Security and Data Analysis
—
Three Case Studies

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

von
Henning Perl
aus
Hannover

Bonn, Februar 2017

Dieser Forschungsbericht wurde als Dissertation von der
Mathematisch-Naturwissenschaftlichen Fakultät der Universität Bonn angenommen
und ist auf dem Hochschulschriftenserver der ULB Bonn
`http://hss.ulb.uni-bonn.de/diss_online` elektronisch publiziert.

1. Gutachter:	Prof. Dr. Matthew Smith
2. Gutachter:	Prof. Dr. Michael Meier
Tag der Promotion:	15.09.2017
Erscheinungsjahr:	2017

Abstract

In recent years, techniques to automatically analyze lots of data have advanced significantly. The possibility to gather and analyze large amounts of data has challenged security research in two unique ways. First, the analysis of Big Data can threaten users' privacy by merging and connecting data from different sources. Chapter 2 studies how patients' medical data can be protected in a world where Big Data techniques can be used to easily analyze large amounts of DNA data. Second, Big Data techniques can be used to improve the security of software systems. In Chapter 4 I analyzed data gathered from internet-wide certificate scans to make recommendations on which certificate authorities can be removed from trust stores. In Chapter 5 I analyzed open source repositories to make predicitions of which commits introduced security-critical bugs. In total, I present three case studies that explore the application of data analysis – "Big Data" – to system security. By considering not just isolated examples but whole ecosystems, the insights become much more solid, and the results and recommendations become much stronger. Instead of manually analyzing a couple of mobile apps, we have the ability to consider a security-critical mistake in *all* applications of a given platform. We can identify systemic errors *all* developers of a given platform, a given programming language or a given security paradigm make – and fix it with the certainty that we truly found the core of the problem. Instead of manually analyzing the SSL installation of a couple of websites, we can consider *all* certificates – in times of Certificate Transparency even with historical data of issued certificates – and make conclusions based on the whole ecosystem. We can identify rogue certificate authorities as well as monitor the deployment of new TLS versions and features and make recommendations based on those. And instead of manually analyzing open source code bases for vulnerabilities, we can apply the same techniques and again consider *all* projects on e.g. GitHub. Then, instead of just fixing one vulnerability after the other, we can use these insights to develop better tooling, easier-to-use security APIs and safer programming languages.

Contents

CHAPTER 1

Introduction

In recent years, techniques to automatically analyze Big Data have advanced significantly. The characteristics of Big Data are often described with the "*three vs*": *velocity* – having lots of data points in quick succession (in units of bytes per second); *variety* – having lots of data of differnt type and/or schemaless data; and *volume* – having a huge amount of information in total (in units of bytes). Even though there is no clear definition of which amount of data qualifies data as Big Data, it is clear that the possibility to gather and analyze large amounts of data has challenged security research in two unique ways. First, the analysis of Big Data can threaten users' privacy by merging and connecting data from different sources. Chapter 2 studies how patients' medical data can be protected in a world where Big Data techniques can be used to easily analyze large amounts of DNA data. Second, Big Data techniques can be used to improve the security of software systems. In Chapter 4 I analyzed data gathered from internet-wide certificate scans to make recommendations on which certificate authorities can be removed from trust stores. In Chapter 5 I analyzed open source repositories to make predicitions of which commits introduced security-critical bugs.

Usable security describes the idea that when designing and evaluating computer security mechanisms, one always has to consider how humans interact with those mechanisms as well. A system that is proven to be secure in theory – assuming correct usage – but at the same time very hard to use will therefore not be secure in practice. As users cut corners and fight the system, the assumptions the security proofs are based on will no longer be satisfied. A seminal work describing this effect in detail is a usability evaluation of PGP called "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." by Whitten and Tygar [1]. The authors showed that even though email encryption with PGP is secure if all rules are followed, the rules too hard to follow and the concepts too hard to grasp that in practice, users just give up. In the same year, Adams and Sasse published

"Users are not the enemy" [2], stating that one should rather blame the system for being too complex instead of the user for not following the rules.

Both works inspired vast research focussing on *end users* interacting with security mechanisms. And indeed many security mechanisms were deemed unusable, such as SSL warnings [3] or passwords [4].

This work focusses on improving the usability of security systems for *IT professionals*. One differentiates between *system administrators*, who will set up and maintain systems that are then used by end users; and *developers*, who write code that is executed by end users. Improving the usability of security mechanisms for these groups holds a great multiplication effect: Every mistake that one developer makes can endanger the data of a multitude of users. Similarly, with good tools, a system administrator is able to protect a multitude of end users.

In the following chapters, this work proposes, implements and evaluates several methods and techniques aiming to gain usability benefits in security and privacy critical contexts through the analysis of large data sets.

Chapter 2 discusses a technique protecting the patients' privacy when conducting research on bio-medical data. The problem domain is as follows: In the future, during medical diagnostics, parts of a patient's DNA may be looked up in a database of DNA samples indicating certain diseases. Doing so unencrypted would leak the patient's DNA. This would result in the database operator knowing which diseases the patent has. Using a homorphically encrypted query (as described in Chapter refchapter:hcrypt) results in a query scheme where the correct result can be retrieved without the database operator knowing the exact query. The big advantage in terms of usability is that this can be completely oblivious to the user (the medical doctor as well as the patient) as well as that it outperforms Private Information Retrieval (PIR) schemes and thus making the approach viable in practice.

Chapter 3 focusses on the evaluation of alternative SSL validation methods. SSL is a technology that touches both system administrator as well as developers. For system administrators, the chore of generating and managing valid certificates is cumbersome and error-prone. A big part of that process is dictated by how the client determines whether or not the certificate is valid. Some alternative validation schemes, such as TACK, put *fewer* burdens on the system administrator, i.e. by not requiring the certificates to be signed at all. Other alternatives, such as AKI or ARPKI, *increase* the burden by offering more configuration options and optionally requiring two or more valid signatures for a certificate. In that regard, it is important to keep in mind the effects the design decisions have for the system administrator. Beyond system administrators, SSL validation holds pitfalls for developers as well. First, the inability to properly validate a SSL certificate opens the application for Man-in-the-middle attacks, accepting all certificates. Second, for applications communicating with a fixed API server, the validation rules could potentially be made stricter by pinning either the particular certificate or the certificate authority signing the concrete certificate. However, pining is still hard to implement correctly in the major programming languages and frameworks.

Whereas the previous chapter is a survey of multiple improvements, Chapter 4 is a concrete contribution to the security of the current CA-based SSL validation system. For this work, I analyzed which certificate authorities never signed a single certificate that actually appears in internet-wide scans. These certificate authorities can thus safely be removed from the browsers' and operating systems' trust stores. This step improves the security of the SSL ecosystem without any drawbacks or usability impacts for either the system administrators, the developers, or the end users.

Finally, Chapter 5 discusses an improvement for the usability of tooling for security experts, precisely code reviewers. With VCCFinder, the tool developed as part of this work, researchers can limit the amount of code they need to review to those parts where security vulnerabilities are most likely.

All in all, each chapter improves on a specific problem statement by considering how the system is interacted with by the users.

CHAPTER 2

Privacy/Performance Trade-off in Private Search on Bio-Medical Data

The trade-off between privacy on the one hand and performance on the other hand is crucial when designing a system that should be both usable as well as secure. This work explores that trade-off in a context where the privacy of the users' data is utterly important: bio-medical data.

This chapter is based on work published as "Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography" at the 2012 IEEE 8th International Conference on E-Science [5], and as "Privacy/performance trade-off in private search on bio-medical data" in the journal Future Generation Computer Systems published by Elsevier [6]. Both publications were joint work with Yassene Mohammed, who consulted on the problem domain of bio-medical data, Michael Brenner, who consulted on the usage of homomorphic cryptography, and Matthew Smith, who did general consulting. The rest is my own work.

2.1 Introduction

Trust remains a crucial challenge concerning the outsourcing of computational tasks to the Cloud in the biomedical domain. Data can be transferred via encrypted channels, but as soon as processing begins, the data is disclosed to the Cloud provider that subsequently has reading access to the data and applies the processing algorithm to it in the clear (c.f. Figure 2.1). This is a significant hindrance for many data intensive applications that could make good use of Cloud computing but have privacy and data protection requirements that forbid disclosure of sensitive data to a third party. The search for DNA or a protein sequence in a database is an example of a biomedical application that uses patient related data that may allow re-identification of the subject. Gymrek et al. [7] describe how the re-identification of subjects is possible using their publicly available DNA data that had been thought of as anonymized.

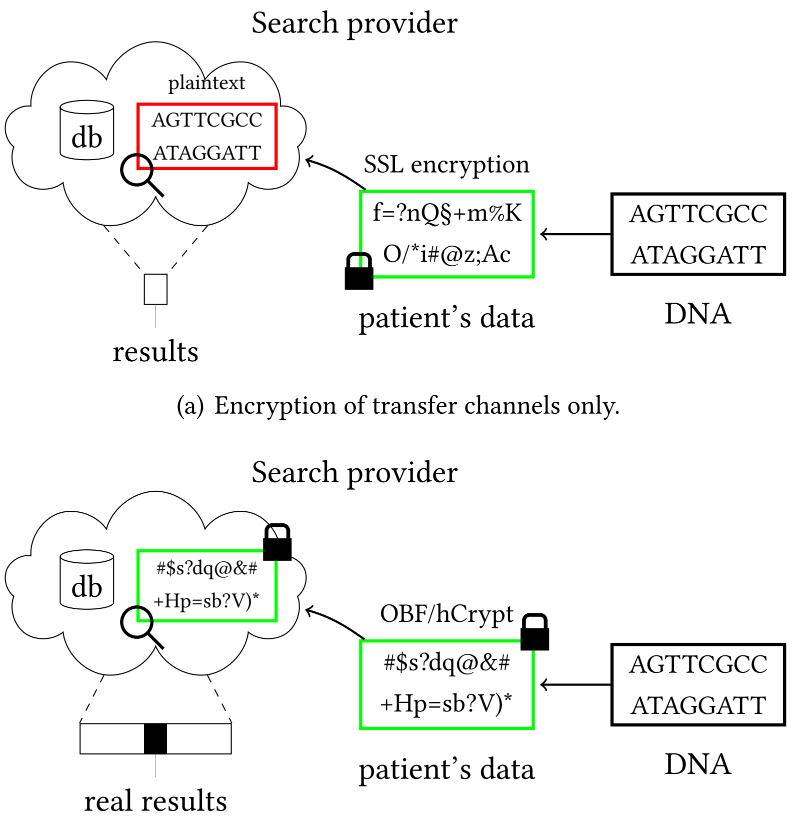
Homomorphic Cryptography (hCrypt, c.f. [8] for Gentry's breakthrough work) is a theoretically promising technology that provides a solution to this problem by allowing a remote party to execute arbitrary algorithms on encrypted data. This would allow the outsourcing of privacy constrained computational tasks to Cloud resources without the need to establish trust, as the Cloud provider could not read the encrypted data. However, the performance of plain hCrypt for large data sets makes it unfeasible for most real world problems if used as a stand-alone solution, since recrypts are necessary after every AND gate and recrypts are a very expensive operation. Even in a recent scheme by Coron, Naccache and Tibouchi [9], a recrypt operation takes 51 s for $\lambda = 62$ bits of security. Thus, the overhead from hCrypt housekeeping quickly dominates the real work.

The family of Private Information Retrieval (PIR) schemes establish a cryptographic protocol that allows a user to search the database without revealing which item was queried. While some progress has been made in terms of runtime and communication complexity by Boneh et al. [10] and Gentry and Ramzan [11], and Kushilevitz and Ostrovsky [12], searching the database is still at best bound linearly to the size of the database. For applications in biology and bioinformatics with large datasets, PIR schemes are still not feasible, as shown by our performance comparison.

2.1.1 Our contributions

Since hCrypt and PIR both solve the problem of private search in theory only, we have developed a Bloom filter based approach to privacy preserving search on databases [5] that allows for the feasible outsourcing of searching a sequence in a database by introducing a privacy/performance trade-off: Instead of a hard security guarantee, the search result is hidden in noise. The ratio between real search and the noise can be configured so that it conforms to data protection regulations. The huge benefit of this somewhat weaker security guarantee is a performance increase of a factor of 2650 for the time it takes to execute a query compared to recent PIR schemes (see Section 2.10.1). In detail, the runtime complexity of our scheme is in $\mathcal{O}(\log |A| + |s| + |R|)$ for a database A with the search term s and the result set R. This qualifies the algorithm to search large data sets.

The search algorithm operates on an Obfuscated Bloom filter (OBF) of the search term. In this chapter we prove that it is cryptographically hard to deduce the search term from the OBF or the result set. Furthermore, the additive property of Bloom filters is used to combine a set of queries into one that matches any search term of the set. This makes



(a) Encryption of transfer channels only.

(b) This article: Obfuscation of the search query.

searching in a stream as efficient as searching in a set of discrete strings. We present a real world application of this principle from the biomedical domain and show that protected and secured search of encrypted DNA sequence queries in the complete human chromosomes is feasible. The output of this Bloom filter based search algorithm is then large enough to conform to the data protection regulations, as well as small enough to allow further processing using hCrypt.

In this article we also present an analysis of how the Bloom filter search performs against private information retrieval schemes on real-world datasets. In performance tests a search of a 50 nucleotides long sequence against human chromosomes can be securely executed in less than 0.1 seconds on a 2.8 GHz Intel Core i7. Finally, we show how the Bloom filter search can be integrated into existing e-Science ecosystems as a Web service. We implement the Bloom filter search as ready-to-use REST service.

2.1.2 Outline

In Section 2.2, we consider alternative methods to solve the problem of privacy preserving search possibilities and evaluate those against the Bloom filter search. In Section 2.3, a common notation for homomorphic encryption schemes and Bloom filters is established. Obfuscated Bloom filters are introduced in Section 2.4 as a crucial component to the search. In Section 2.5, we discuss the Bloom filter search in detail and show a security analysis in Section 2.6. An implementation of the algorithms along with Web service interfaces is introduced in Section 2.8; a real world use case is analyzed in Section 2.9. We conclude with a performance evaluation and comparison with two PIR schemes in Section 2.10.

2.2 Related Work

2.2.1 Private Information Retrieval (PIR)

In 2007, Boneh et al. [10] published a protocol that supports queries on encrypted data using a somewhat homomorphic crypto system by Boneh, Goh and Nissim [13]. This crypto system allows for an arbitrary amount of additions, but only one multiplication. The system they present has a communication complexity of $O(\sqrt{A}\log^3 A)$ with A being the size of the database, whereas our approach only has a communication complexity of $\mathcal{O}(|s|)$ with s as the search term, as shown in Equation (2.5). Therefore the communication complexity of our scheme does not depend on the size of the database at all. Boneh et al. achieve a runtime complexity of $\mathcal{O}(|A| \cdot |s| \cdot \operatorname{poly} \lambda)$ in combination with the PIR protocol by Cachin, Micali and Stadler [14]. In comparison, the total runtime complexity of our scheme is in $\mathcal{O}(\log |A| + |s| + |R|)$ for $|R|$ being the size of intermediate results produced by the Bloom filter search. Comparing security properties, Boneh's scheme completely hides the query from the server, whereas our scheme hides the query in $|R| = \binom{\lambda+k}{\lambda}$ queries (where k is the number of hash functions used in the Bloom filters and λ the obfuscation parameter). When setting $|R| \approx |A|$, we achieve the same security and runtime complexity as [10]. The parameter λ can be used to control the level of obfuscation and enables a security/performance trade off. This allows for the practical use of the system for a given amount of obfuscation (c.f. Sections 2.4, 2.9).

Camenisch, Dubovitskaya and Neven [15] show how to integrate access permissions and policies into an oblivious transfer protocol by introducing a third party. This work focuses on confidential access to public databases without access policies.

Canetti, Riva and Rothblum [16] determine ways to provide verifiable results of delegated computation and information retrieval. Apart from the performance problem, this is another great challenge for future delegation scenarios but out of the scope of this work.

2.2.2 Garbled Circuits

The first secure approach to solve the problem of secret function evaluation was introduced in 1986 by Yao [17]. He presented the concept of *Garbled Circuits*, in which algorithms are translated to single-pass boolean circuits. The state tables of the gates in the circuit are then encrypted and shuffled within the state table, disguising the boolean function of a gate and essentially the function of the whole circuit.

Although this method is usable to some extent in an implementation by Malkhi et al. [18] and Malka [19], the core concept has some inherent deficits:

- The encryption and shuffling of the state tables introduce a dependency between the gates. This means that the circuit is only able to pass in one large pre-defined run. Modularity is not possible.
- Furthermore, the security of the scheme relies on the fact that each gate only runs once. Therefore, only linear circuits are possible. Loops, for example, have to be unrolled completely.
- Lastly, the input from the circuit creator is encoded right into the garbled circuit. In consequence, no memory access is possible, because the value of a memory cell can not act as input for the next cycle.

These restrictions lead to the fact that only limited algorithms can be transformed into a garbled circuit. Since the circuits can only be passed once, it is impossible to search twice with the same database and circuit.

2.2.3 Trusted Computing

In *Trusted Computing* [20], the integrity of the system is ensured through specialized hardware, the Trusted Platform Module (TPM). The TPM can measure the integrity of the bootloader, the operating system and software running in the operating system and help prevent tampered components from starting. The data and programs processed by the platform cannot be protected against the owner of the platform, as they are in possession of the root key for the platform. Therefore, Trusted Computing can not be used to outsource searches if the database server is not trusted.

2.2.4 Encrypted CPU / hCrypt

Gentry [8, 21] introduced a fully homomorphic crypto system. In this work he also stated that it is possible to evaluate circuits with arbitrary depth by mapping the mathematical operations to boolean operations, thus offering significant ground work for the encrypted execution of arbitrary programs. Based on this and the cryptosystem by Smart and Vercauteren [22], Brenner et al. [23] developed an encrypted CPU capable of executing arbitrary programs by addressing issues such as memory access, branching and self-modifying code. With this CPU it is possible to execute arbitrary encrypted code as well as access an encrypted memory from within the code.

Although the simulated CPU solves the problem of secure function evaluation (SFE), it is still slower than native code by several orders of magnitude. The following reasons can be identified:

- Because of the oblivious memory access, the simulated CPU must re-evaluate all memory cells in every cycle, regardless of whether or not the plaintext value actually changed.
- It is not possible to determine the end of a program without knowledge of the secret key. Therefore when approximating the number of required cycles for the encrypted CPU, worst-case runtime has to be assumed.

2.3 Preliminaries

In this section we will introduce the basic components that are part of our search scheme as well as highlight their utility in the scheme.

2.3.1 Homomorphic Encryption Schemes

The discovery of a fully homomorphic encryption scheme by Gentry [8, 21] in 2009 is an important foundation for the techniques described in this work. However, for the construction of our approach we do not require a particular instance of a homomorphic encryption scheme. Thus we use this generic definition:

Definition 2.3.1. A *fully homomorphic encryption scheme* $\mathcal{E}$ is a tuple of functions

$$\begin{aligned}(\mathsf{KeyGen}_{\mathcal{E}}() &\mapsto (pk, sk),\\ \mathsf{Encrypt}_{\mathcal{E}}(m, pk) &\mapsto \mathfrak{c},\\ \mathsf{Decrypt}_{\mathcal{E}}(\mathfrak{c}, sk) &\mapsto m,\\ \mathsf{Evaluate}_{\mathcal{E}}(\mathfrak{c}_i, C, pk) &\mapsto \mathfrak{c}')\end{aligned}$$

with the following properties:

1. *Correctness of the encryption scheme:*

$$\mathsf{Decrypt}(\mathsf{Encrypt}(m, pk), sk) = m$$

   for all outputs (pk, sk) of KeyGen().

2. *Correctness of the homomorphic property:*

$$\begin{aligned}&\mathsf{Decrypt}(\mathsf{Evaluate}(\\ &\qquad(\mathsf{Encrypt}(m_0, pk), \ldots, \mathsf{Encrypt}(m_n, pk)), C, pk), sk)\\ &\qquad\qquad = C(m_0, \ldots, m_n)\end{aligned}$$

   for all outputs (pk, sk) of KeyGen() and all boolean circuits $C : \{0, 1\}^m \to \{0, 1\}$.

For the remainder of this work we assume that all functions (KeyGen() ...) belong to the same encryption scheme, even without the index $\mathcal{E}$.

We set the plain text space P to be $\{0, 1\}$ and treat the cipher texts as encrypted bits for the boolean operations XOR and AND (which correspond to the addition and multiplication of two cipher texts). The cipher text space C of course depends on the particular scheme used.

In the search scheme, homomorphic encryption is used to execute an exact match search over the query and a reduced set of the database. Note that since the homomorphic encryption scheme is an *asymmetric* scheme, someone that wants to evaluate a circuit only needs the public key pk (along with the inputs) to compute Evaluate(). Further, since Encrypt() can also be executed by the evaluator, the computations can also consist of a mix of plaintext and ciphertext. The result would then be encrypted and retrievable only through the secret key sk.

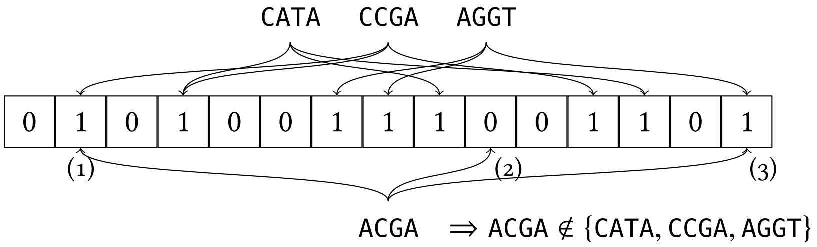


Figure 2.1: Example of a Bloom filter for {CATA, CCGA, AGGT} with three hash functions.

2.3.2 Bloom Filters

Bloom filters have been proposed by Bloom [24] in 1970. They provide a way to check whether a string is included in a set of strings, with a small probability that the Bloom filter wrongly outputs that the string is in the set while it is not (false positive). A Bloom filter is a bit array with a fixed length m. In order to create a Bloom filter from strings, one needs to also agree on a set of k hash functions that each have a target set corresponding to the indexes of the Bloom filter $\{0, \dots m\}$. For example, a Bloom filter for a string x is then created by hashing the string with each of the k hash functions and marking that index in the Bloom filter by writing a 1 at the position of the index. A Bloom filter for a set of strings $\{a, b, c\}$ is created by first creating the Bloom filters of the elements a, b, and c and then computing a component-wise boolean OR, i.e. a position in the Bloom filter is marked if that position was marked in any of the elements' Bloom filters. The creation of Bloom filters with size $m = 15$ and $k = 3$ hash functions for the set {CATA, CCGA, AGGT} is shown in Figure 2.1.

In order to use the Bloom filters to check for set membership (e.g. whether AGGT $\in$ {CATA, CCGA, AGGT}), the Bloom filters of an element AGGT and a set {CATA, CCGA, AGGT} are checked component-wise. If an index exists in the Bloom filters that is marked in the Bloom filter of the element AGGT but not in the Bloom filter of the set, then certainly AGGT is not included in the set (because the Bloom filter of the set is just a component-wise OR over the elements). If, however, every marked position in the Bloom filter of AGGT is also marked in the Bloom filter of the set, then AGGT is probably included in the set. This probability is bound by $(1 - e^{-k(|A|+0.5)/(m-1)})^k$ as shown by Goel and Gupta [25]. Going back to the example in Figure 2.1, the string ACGT is not included in {CATA, CCGA, AGGT}, because hash function (2) for ACGT marked a position that was not marked for either CATA, CCGA, or AGGT.

Notation We denote

- $\mathcal{B}^{k,m}(a)$ for the Bloom filter of a with size m and k hash functions,

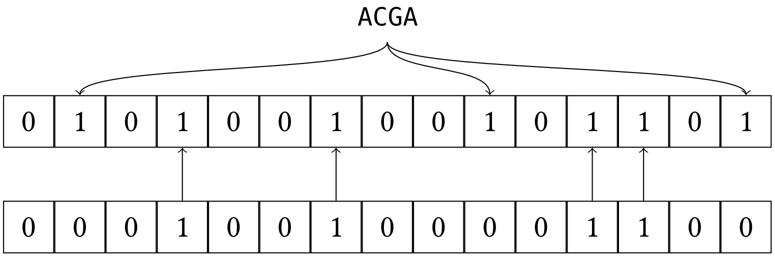


Figure 2.2: A Bloom filter for ACGA with obfuscation parameter $\lambda = 4$.

- $\mathcal{B}^{k,m}(A)$ for the Bloom filter of a set A,
- $\mathcal{B}^{k,m}(a) \in \mathcal{B}^{k,m}(A)$ if for all indices i: $\mathcal{B}^{k,m}(a)_i \leq \mathcal{B}^{k,m}(A)_i$; and $\mathcal{B}^{k,m}(a) \notin \mathcal{B}^{k,m}(A)$ otherwise. Note that we use '≤' here instead of '=' to compare the values of the index positions, because this way the definition naturally extends to counting Bloom filters.

In this work, Bloom filters are used to construct a binary tree to quickly reduce a large set of data to a small set of possible matches via a binary search. In order to search for a string x in a large database, we recursively check for set membership of x in the set containing all words in the database and continue to check for membership in the left and right half of the whole set if x is included in it.

2.4 Obfuscated Bloom Filters

We extend Bloom filters to make false positives in the set membership check more probable. The motivation behind this is to pass an Obfuscated Bloom filter (OBF) of the query to the database that then finds all database entries that contain the OBF. Without obfuscation, this set would be small enough for the database to learn about the actual query. By using the OBF, we will show that the probability to retrieve the plain Bloom filter can be made small enough that the actual result is hidden in the large set of false positives, i.e. the noise; yet small enough to make the processing of these intermediate results in the homomorphic cipher text space feasible. Thus it can be ensured that only a configurable percentage of the results is not noise.

Obfuscating a Bloom filter works by incrementing λ random components of the Bloom filter $\mathbf{b} = \mathcal{B}^{k,m}(a)$, where λ is the obfuscation parameter. Figure 2.2 shows a Bloom filter for a string ACGA that is obfuscated with $\lambda = 4$. The hamming weight of the OBF is always $k + \lambda$.

The following algorithm increments the components by picking a random index until one is found who's entry not zero, and then setting that location to one.

```
function obfuscate($\mathbf{b}$, $m$, $\lambda$):
    for $l$ in $[\lambda]$ {
        while ($i \xleftarrow{R} [m]$ and $b_i = 1$)
        $\mathbf{b}_i \leftarrow 1$
    }
```

Because obfuscation marks λ additional positions in the Bloom filter, the set membership check needs some modification. Recall that set membership is tested by checking for each position in the Bloom filter of the potential element whether that position is also marked in the set. Now that the potential element is obfuscated, set membership is already concluded if at least k marked positions exist that are also marked in the set.

We now show that OBFs preserve the set membership property of plain Bloom filters.

Claim 1. Given a Bloom filter $\mathbf{B} = \mathcal{B}^{k,m}(A)$ for a set A and a (plain) Bloom filter $\mathbf{b} = \mathcal{B}^{k,m}(a)$ as well as an obfuscated version $\mathbf{b}''' = \text{obfuscate}(\mathbf{b}, m, \lambda)$. Then:

1. $\mathbf{b} \in \mathbf{B} \Rightarrow \mathbf{b}' \in_\lambda \mathbf{B}$
2. $\mathbf{b}' \in_\lambda \mathbf{B} \Rightarrow \Pr[\mathbf{b} \in \mathbf{B}] = 1/\binom{k+\lambda}{k}$

Proof. Let $\mathbf{v} \leftarrow \mathbf{b}' - \mathbf{b}$, $\mathbf{v} \in \Delta^\lambda$ be the vector the Bloom filter $\mathbf{b}$ was obfuscated with.

1.

$$\begin{aligned} \mathbf{b} \in \mathbf{B} &\Rightarrow \forall i \in [m] : \mathbf{b}_i \leq \mathbf{B}_i \\ &\Rightarrow \forall i \in [m] : \mathbf{b}'_i - \mathbf{v}_i \leq \mathbf{B}_i \\ &\Rightarrow \forall i \in [m] : \mathbf{b}'_i \leq \mathbf{B}_i + \mathbf{v}_i \\ &\Rightarrow \forall i \in [m] : \exists \mathbf{v} \in \Delta^\lambda : \mathbf{b}'_i \leq \mathbf{B}_i + \mathbf{v}_i \\ &\Rightarrow \mathbf{b}' \in_\lambda \mathbf{B} \end{aligned}$$

2. Given $\mathbf{b}' \in \mathbf{B}$, $\Pr[\mathbf{b} \in \mathbf{B}] = \Pr[\mathbf{b}' - \mathbf{v} \in \mathbf{B}]$ for the random vector $\mathbf{v}$ that $\mathbf{b}$ was obfuscated with. For a fix obfuscated vector $\mathbf{b}'$ (with $\sum_i \mathbf{b}'_i = k + \lambda$), there exist $\binom{k+\lambda}{\lambda}$ vectors $\mathbf{v}$ so that $\mathbf{v} = \mathbf{b}' - \mathbf{b}$. Therefore $\Pr[\mathbf{b} \in \mathbf{B}] = 1/\binom{k+\lambda}{\lambda}$.

□

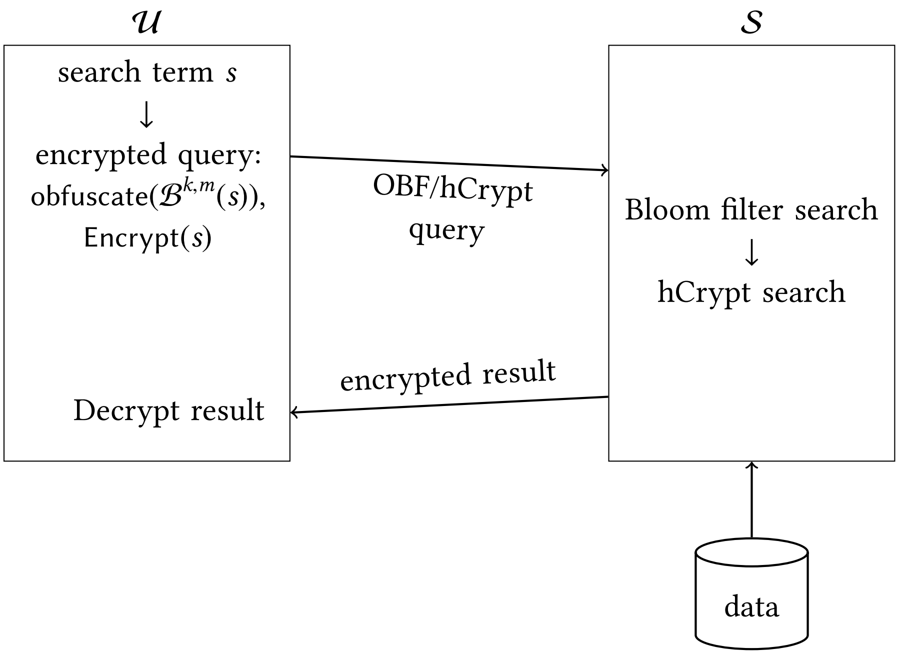


Figure 2.3: Model for the search query.

2.5 Bloom Filter Search

2.5.1 High-Level View of the Search Scheme

Figure 2.3 shows a general overview of how a search query is created and evaluated. User $\mathcal{U}$ wants to query Server $\mathcal{S}$ for a search term s without $\mathcal{S}$ learning about this search term or the result. First of all, $\mathcal{U}$ and $\mathcal{S}$ agree on a common alphabet Σ that influences the following parameters:

- The domain of each hash function used in the Bloom filters is Σ^*.
- The search term s of $\mathcal{U}$ is an element of Σ^* (a word of the alphabet Σ).
- The database A of $\mathcal{S}$ is a subset of $\mathcal{P}(\Sigma^*)$, the power set of Σ^* (a set of words of the alphabet Σ).

To execute a search, $\mathcal{U}$ transforms the *search term* into an *encrypted query*, which is an OBF of s.

This query is then sent to $\mathcal{S}$ and used in the *search algorithm*. The server uses the OBF to reduce the database to a smaller set of possible matches.

We formalize this protocol as follows.

1. *Setup of the Bloom filter tree.*

$\mathcal{S}$ sets $T \leftarrow$ buildTree(root, database).

2. *Construction of the query.*

   The user $\mathcal{U}$ chooses a search term s and an hCrypt key pair $(pk, sk) \leftarrow$ KeyGen().

   He sets $\mathbf{b} \leftarrow$ obfuscate($\mathcal{B}^{k,m}(s), \lambda$) as well as $\mathfrak{s} \leftarrow$ Encrypt(s, pk).

3. *Transfer of the encrypted query.*

   $\mathcal{U}$ transfers $(\mathbf{b}, \lambda, \mathfrak{s}, pk)$ to $\mathcal{S}$.

4. *Bloom filter search.*

   $\mathcal{S}$ obtains a set of results $R \leftarrow$ searchTree($T, \mathbf{b}, \lambda$).

5. *hCrypt Search.*

   $\mathcal{S}$ obtains an encrypted result $\mathfrak{res}$ of the hCrypt algorithm on R and $\mathfrak{s}$.

6. *Transfer of the results.*

   $\mathcal{S}$ transfers $\mathfrak{res}$ back to $\mathcal{U}$, who then retrieves the final result $r \leftarrow$ Decrypt($\mathfrak{res}, sk$).

2.5.2 Setup of the Bloom Filter Tree

As mentioned above, a binary tree of Bloom filters is used to pre-filter the search results. Every node of the binary tree is a tuple $(\mathbf{b}, d, l, r)$, where $\mathbf{b}$ contains the Bloom filter associated with this node and l and r reference the left and right child of the node respectively. The l and r references are set to 'null' iff. the node is a leaf. In this case d holds the database entry associated with the leaf.

Now we can describe the algorithm that builds the binary tree from a database A. Note that this step is independent of the actual query and can therefore be calculated beforehand and used for many different queries.

```
function buildTree($node, A$):
    $node.b \leftarrow \mathcal{B}^{k,m}(A)$
    if ($|A| = 1$) {
        $node.l \leftarrow$ null
        $node.r \leftarrow$ null
        return
    }
    buildTree($node.l, A|_{0..\lfloor |A|/2 \rfloor}$)
    buildTree($node.r, A|_{\lfloor |A|/2 \rfloor +1..|A|}$)
    return
```

The algorithm calculates the Bloom filter for the current node and then recursively calls itself on the left and right half of the database A when filling the left and right children

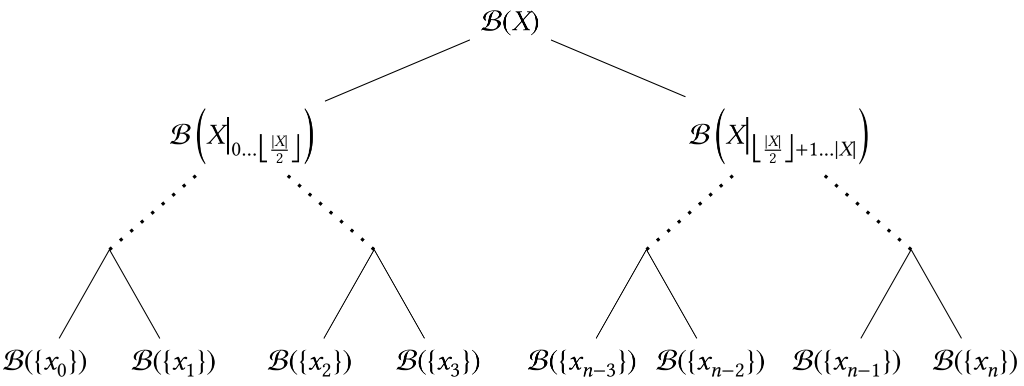

$\mathcal{B}(X)$
$\mathcal{B}\left(X|_{0\ldots\lfloor\frac{|X|}{2}\rfloor}\right)$
$\mathcal{B}\left(X|_{\lfloor\frac{|X|}{2}\rfloor+1\ldots|X|}\right)$
$\mathcal{B}(\{x_0\})$
$\mathcal{B}(\{x_1\})$
$\mathcal{B}(\{x_2\})$
$\mathcal{B}(\{x_3\})$
$\mathcal{B}(\{x_{n-3}\})$
$\mathcal{B}(\{x_{n-2}\})$
$\mathcal{B}(\{x_{n-1}\})$
$\mathcal{B}(\{x_n\})$

Figure 2.4: Binary tree of Bloom filters

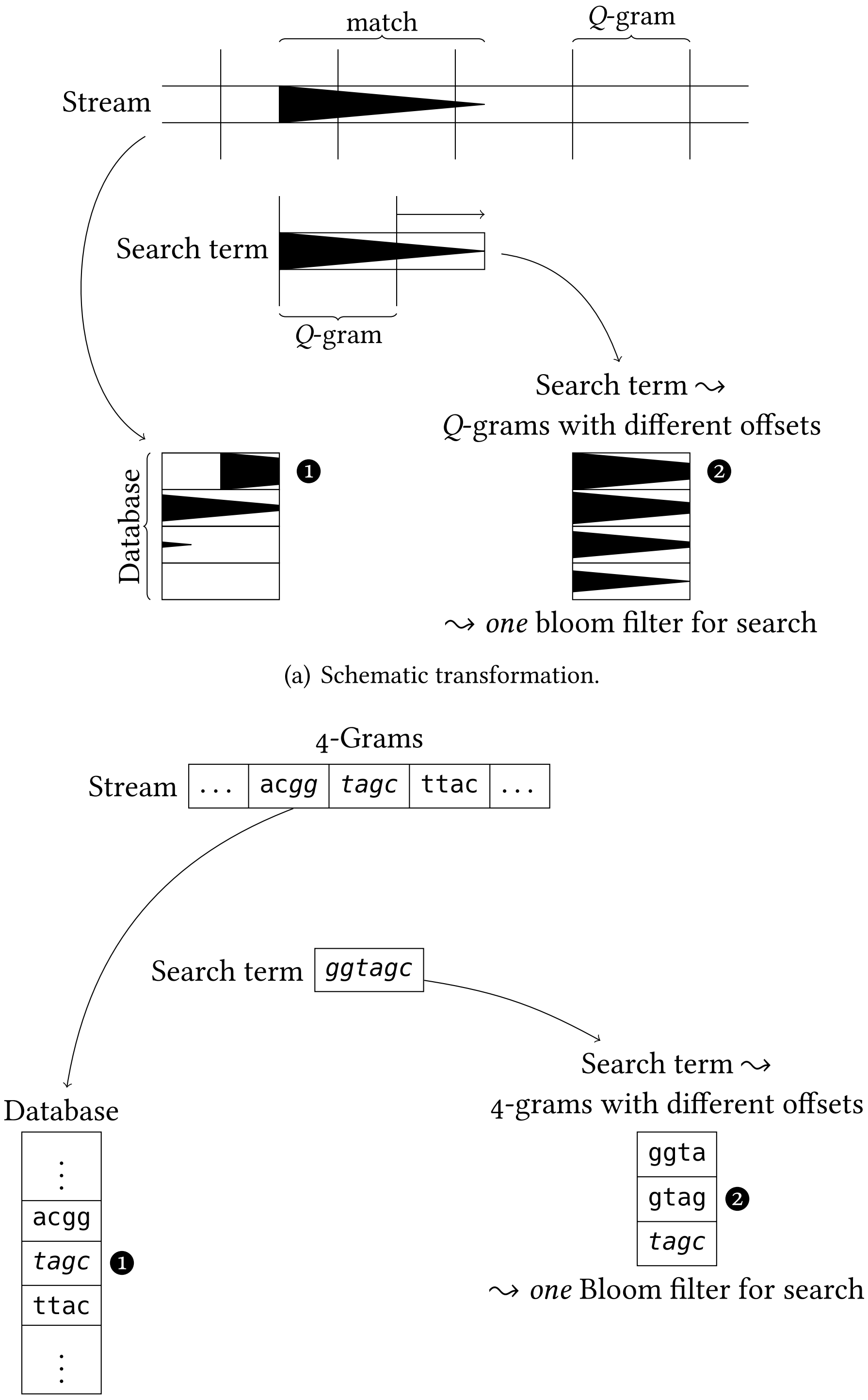
of the node. Note that although the initial construction of the binary tree runs in time $O(|A|)$, updates to the binary tree are quite cheap, because only the nodes along the path from the changed node to the root need to be updated. Updates therefore run in time $O(\log|A|)$.

Figure 2.4 shows the content of the binary tree after the construction.

2.5.3 Search Using Bloom Filters and Binary Search

Once the binary tree is constructed, the Bloom filter search is just a modified binary search over the Bloom filter of the query. In the following algorithm, *node* is initially the root node of the binary tree and $b = \text{obfuscate}(\mathcal{B}^{k,m}(s), \lambda)$ is the OBF of the query.

```
function searchTree($node, b, \lambda$):
    if ($b \notin_\lambda node.b$) {
        return $\emptyset$
    }
    if ($node.l = \text{null}$ or $node.r = \text{null}$) {
        return $\{node.d\}$
    } else {
        $l \leftarrow \text{searchTree}(node.l, b, \lambda)$
        $r \leftarrow \text{searchTree}(node.r, b, \lambda)$
        return $l \cup r$
    }
```



(a) Schematic transformation.

(b) Transformation by example for the query “ggtagc”.

Figure 2.5: Transforming Bloom filter search to a stream search.

Extension to Stream Search

For our application scenario we require the capability to search on streams, so we show how the Bloom filter search can be modified to find a substring in a stream (instead of a discrete set of words). Let s be the search term of $\mathcal{U}$ and A the database of $\mathcal{S}$, i.e. a character stream. The following steps are necessary to achieve a stream search:

1. Set the length of the Q-grams $Q \leq \lfloor |s|/2 \rfloor$. This guarantees that at least one Q-gram consists only of characters found in the search term.
2. Split the stream into Q-grams, thus getting a database of roughly $|A|/Q$ words of size Q (1. in Figure 2.5).
3. From the search term, generate $|s|$ Q-grams with offsets $0 \ldots |s| - 1$ (2. in Figure 2.5).
4. Combine the search term Q-grams into a single Bloom filter by adding them componentwise.
5. Execute the search with one Bloom filter as described above.
6. For each result, do an exact-match comparison in the cipherspace, working directly on the encrypted search term and the stream.
7. Compress the result in the cipherspace to return the encrypted positions of the search term in the stream.

This stream search extension is what we use to implement the real world use case in Section 2.9.

2.5.4 Exact Search Using hCrypt

After $\mathcal{S}$ executed the Bloom filter search described above, the (intermediate) set of results R contains matches based on the OBF of the original search term s and may contain only a small percentage of real results. In order to send just these real results back to the user $\mathcal{U}$, hCrypt can optionally be used to execute an exact match search. This serves as an example of how hCrypt can be used on the results of the Bloom filter search to add further processing to the whole search algorithm.

The goal of this step is for $\mathcal{S}$ to construct a single encrypted result $\mathfrak{res}$ that contains a small number of matches of the search query of $\mathcal{U}$. First, $\mathcal{S}$ constructs an indication vector $\mathfrak{ind}$ that marks all exact matches. Then, $\mathcal{S}$ uses this indication vector to filter just those intermediate results into the final set of results which were marked before.

As all these transformations happen in the homomorphic cryptospace, $\mathcal{S}$ gains no knowledge over the final results during its construction.

Exact Match Search

First, $\mathcal{S}$ marks which results are actual matches of s in the homomorphic cipherspace. This is done by generating an encrypted bit-array $\mathfrak{ind}$ with the same size as the set of results that will serve as an indication vector. More precisely let $\{r_i\}_{i=1}^{l} = R$ be the set of results. Then we want to construct an encrypted indication vector $\{\mathfrak{ind}_i\}_{i=1}^{l} \in C^l$ for the cipherspace C so that

$$r_i = s \Leftrightarrow \mathsf{Decrypt}(\mathfrak{ind}_i, sk) = 1 \quad \text{for all } i \in [l].$$

Note that in order to construct such a $\mathfrak{ind}$ in the homomorphic cipherspace, $\mathcal{S}$ only needs an encrypted version of s as well as the public key pk, which both can be sent to $\mathcal{S}$ as part of the encrypted query.

We assume that the plain text space $\mathcal{P} = \{0, 1\}$. In order to compare words $\in \Sigma^*$ in the ciphertext space, words from the alphabet must first be transformed to a binary representation. Let $\mathsf{binary} : \Sigma^* \to \{0, 1\}^*$ be this function. This function can easily be constructed by first giving each character $c \in \Sigma$ a unique number and then encode a word $w \in \Sigma^*$ as a sequence of the padded binary representation of each character in w.

Let

$$\mathfrak{s} = \{\mathfrak{s}_j\}_{j \in [|\mathfrak{s}|]} \leftarrow \{\mathsf{Encrypt}(\mathsf{binary}(s)_j, pk)\}$$

be the encrypted search term and

$$\mathfrak{b} = \{\mathfrak{b}_{ij}\}_{i \in [l], j \in [|\mathfrak{s}|]} \leftarrow \{\mathsf{Encrypt}(\mathsf{binary}(r_i)_j, pk)\}$$

be the encrypted intermediate set of results. The circuit for the character-level comparison is constructed as follows:

$$\mathfrak{ind}_i \leftarrow \bigwedge_j \mathfrak{b}_{ij} \oplus \mathfrak{s}_j \oplus 1 \quad \text{for all } i \in [l]. \tag{2.1}$$

Now that the indication vector is filled, the marked intermediate results need to be mapped to the final result.

Compressing the results

Assume that there are at most c entries in the vector $\mathfrak{ind}$ marked with 1. This number can be approximated from the number of results in the result set and the security parameter λ. Then an indication vector containing at most c marks can be expressed as a bit-vector with length $c \cdot |\mathfrak{s}|$. The basic idea is to shift each result which is a match (for which

$$\overbrace{\begin{pmatrix}\text{AGCT}\\\text{AGGT}\\\text{CGAA}\\\text{TGAG}\\\text{AGGT}\\\text{GGAT}\\\text{AGGT}\end{pmatrix}}^{R}\quad\overbrace{\begin{pmatrix}0\\1\\0\\0\\1\\0\\1\end{pmatrix}}^{\mathfrak{ind}}\rightsquigarrow\overbrace{\begin{pmatrix}0&0\\0&1\\0&0\\0&0\\1&0\\0&0\\1&1\end{pmatrix}}^{\mathfrak{sum}}\rightsquigarrow\overbrace{\begin{pmatrix}0&0&0\\0&0&1\\0&0&0\\0&0&0\\0&1&0\\0&0&0\\1&0&0\end{pmatrix}}^{\mathfrak{mask}}\rightsquigarrow\underbrace{\begin{pmatrix}\text{encrypt(AGGT)}\\\text{encrypt(AGGT)}\\\text{encrypt(AGGT)}\end{pmatrix}}_{\mathfrak{res}}$$

Figure 2.6: Steps for compressing a set of results. For the query "AGGT" the exact-match search circuit marked entries 2, 5 and 7 in $\mathfrak{ind}$, which is then transfomed according to Eq. (2.2)–(2.4).

$\text{Decrypt}(\mathfrak{ind}_i, sk) = 1$) by $|s|$ bits.

Let $\text{hamming}(v)$ be the circuit that calculates the hamming weight of v as constructed by Smart and Vercauteren [22, p. 15]. This circuit returns a bit vector of length $\lceil \ln c + 1 \rceil$ if there are at most c 1's in v. Then, write down the cumulative sum of $\mathfrak{ind}$ as the $|R| \times \lceil \ln c + 1 \rceil$-matrix $(\mathfrak{sum}_{ij})$ using the following circuit:

$$\mathfrak{sum}_{ij} = \text{hamming}(\mathfrak{ind}_{0..i})_j \wedge \mathfrak{ind}_i \tag{2.2}$$

Next, set the $|R| \times c$-matrix

$$\mathfrak{mask}_{ij} = \begin{cases} 1 & \sum_{k=0}^{\lceil c \rceil} 2^k \cdot \mathfrak{sum}_{ik} = j \\ 0 & \text{else.} \end{cases} \tag{2.3}$$

The final result can then be written as a $c \times |R|$-matrix $(\mathfrak{res}_{ij})$ with

$$\mathfrak{res}_{ij} = \bigoplus_{k=0}^{|R|-1} \text{encrypt}(R_k)_j \wedge \mathfrak{mask}_{ki}. \tag{2.4}$$

Figure 2.6 shows the steps for a small example. The rows are mirrored for better readability.

After this step, $\mathcal{S}$ holds an encrypted final set of results $\mathfrak{res}$, which contains at most c matches of the database against the search term s. These results are then sent to $\mathcal{U}$, who decrypts $\mathfrak{res}$ using her private key sk.

2.6 Security Analysis

For the search scheme to be secure one needs to show that an adversary is unable to extract the plain search query from the inputs, i.e. security against ciphertext-only attacks (COA) on the encrypted query. More precisely, given a tuple

$$q = (\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(s)), \lambda, \mathsf{Encrypt}(s, pk), pk)$$

it is hard for an adversary $\mathcal{A}$ to retrieve s in probabilistic polynomial time (PPT). With that in mind we define the following security game:

Definition 2.6.1. For COA-security of the search scheme, we consider the following game between a challenger C (which takes the role of the user in our scheme) and an adversary $\mathcal{A}$. The functions $\mathsf{KeyGen}()$, $\mathsf{Encrypt}()$, $\mathsf{Decrypt}()$ and $\mathsf{Evaluate}()$ are assumed to be indistinguishable under chosen-plaintext attack (IND-CPA secure). For example, the encryption schemes by Brakerski, Gentry and Vaikuntanathan [26] or Smart and Vercauteren [22] are all IND-CPA secure. The game consists of the following steps:

1. C chooses a random s, generates a key pair $(pk, sk) = \mathsf{KeyGen}()$, computes q as above and sends it to $\mathcal{A}$.
2. $\mathcal{A}$ can repeatedly ask for the encryption of a specific query l; C replies by sending $(\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(l)), \lambda, \mathsf{Encrypt}(l, pk), pk)$.
3. $\mathcal{A}$ computes a query s' and sends it to C.
4. C outputs 1 iff. $s = s'$.

The adversary's advantage $\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is defined as $\Pr[s = s']$. A search scheme is COA-secure if, for all $\mathcal{A} \in PPT$ the function $\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is negligible.

Claim 2. $\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is a negligible function for any adversary that runs in probabilistic polynomial time.

Proof. We show that $\mathrm{Adv}_{\mathcal{A}}(\lambda) \leq 1/\binom{k+\lambda}{\lambda}$, which is negligible in λ, by contradiction.

Assume that there exists an adversary $\mathcal{A}$ that runs in probabilistic polynomial time and breaks the security game with advantage $> 1/\binom{k+\lambda}{\lambda}$. Then we distinguish between three distinct cases:

1. $\mathcal{A}$ breaks the game by using $(\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(s)), \lambda)$ to retrieve s from the encrypted query $(\mathsf{Encrypt}(s, pk), pk)$.

   $\mathcal{A}$ can then be used to break IND-CPA security of the underlying crypto system. Recall that the IND-CPA security game challenges an adversary $\mathcal{A}'$ who generated two plaintexts (s_1, s_2) to decide whether a ciphertext c is an encryption of s_1 or s_2 with non-negligible advantage. This is how $\mathcal{A}'$ breaks the game using $\mathcal{A}$:

a) $\mathcal{A}'$ sends (s_1, s_2) to the challenger and receives a ciphertext c and a public key pk.

b) $\mathcal{A}'$ uses $\mathcal{A}$ to ask for the encrypted queries $q_i = \{(\mathbf{b})_i, \lambda, c_i, pk'\}_{i=1}^{2}$.

c) $\mathcal{A}'$ sets $q_i \leftarrow \{(\mathbf{b})_i, \lambda, c, pk\}_{i=1}^{2}$ and uses $\mathcal{A}$ to retrieve s_1', s_2' (which we assumed $\mathcal{A}$ can do).

d) $\mathcal{A}$ outputs i where $s_i' = s_i$.

This contradicts the assumption that the underlying cryto system is IND-CPA secure.

2. $\mathcal{A}$ already breaks the game with inputs only $(\mathsf{Encrypt}(s, pk), pk))$.

   It is easy to see that this again gives an adversary $\mathcal{A}'$ that breaks CPA-security of the underlying homomorphic crypto system and therefore contradicts the assumption that the crypto system is IND-CPA secure. Note that this also covers the case where $\mathcal{A}$ uses $(\mathsf{Encrypt}(s, pk), pk))$ to retrieve s from $(\mathbf{b}, \lambda)$.

3. $\mathcal{A}$ already breaks the game with inputs only $(\mathbf{b}, \lambda)$.

   $\mathcal{A}$ can then construct a large set $A = \{a_i\}_{i=1}^{l}$ so that $\mathbf{b} \in_\lambda \mathcal{B}^{k,m}(a_i)$ for each i. Then starting from the assumption

$$\frac{1}{\binom{\lambda+k}{\lambda}} < \mathrm{Adv}_{\mathcal{A}}(\lambda)$$

   we have the contradiction

$$\begin{aligned}
\mathrm{Adv}_{\mathcal{A}}(\lambda) &= \Pr[s = a_i] && \text{(Definition of } \mathrm{Adv}_{\mathcal{A}}\text{)} \\
&\leq \Pr[\mathbf{b} \in_\lambda \mathcal{B}^{k,m}(a_i)] && \text{(Apply Bloom filters)} \\
&= \frac{1}{\binom{\lambda+k}{\lambda}} && \text{(Claim 1)} \\
\Rightarrow \frac{1}{\binom{\lambda+k}{\lambda}} &< \frac{1}{\binom{\lambda+k}{\lambda}}\,.
\end{aligned}$$

□

2.7 Choosing an Obfuscation Parameter

The real results to noise ratio is a measure of how unlikely captured data by an attacker would be useful. The smaller the ratio is, the harder an attacker can make any use of it. For instance hiding in 1 % means the probability that any conclusions of an attacker about the original search term and the results will be wrong in 99 % of the cases. So if a study associate one sequence S with a specific illness and we search for S on the genome with 1 % real results to noise hiding ratio, it means that an attacker will have 99 wrong sequences along with S. An attacker can increase the certainty of her choices by including more reasoning knowing for instance which searches the user is interested in, what the user is attempting to publish, what results does not make sense for the user, and so on. Anyhow, taking into account that the real results to noise hiding ratio can be variable and is unknown to the attacker, it involves huge effort from an attacker to reach a certainty that make any conclusions statistically valid.

2.8 Implementation

2.8.1 Source Code

The Bloom filter search algorithm was implemented in C as a binary tree with one root node and a pointer to the left and right child of the node. Each node contains information about the Bloom filter that represents the current subtree. Further, each leaf contains the corresponding database value.

The construction of the Bloom filter tree (i.e., indexing of the database) is done iteratively from a file. The algorithm scans the file to determine the number of nodes required and then builds the Bloom filter tree from the bottom up. The advantage over a naïve recursive implementation is that all memory allocation can be done in one call.

The source code can be found at `https://github.com/hperl/bf-search` and builds using CMake. For compilation and linking, we require OpenSSL (for the hash functions in the Bloom filter tree) as well as Ruby 1.9 for the Web service (described below). Amongst others, the command-line program `bloom_search` is built, which takes as a mandatory parameter a chromosome file to index. In the path `/chromosomes` is a script `download.sh` that fetches all human chromosome files from `http://hgdownload.cse.ucsc.edu/`.

The implementation code for the Bloom filter search is organized in four parts: The main executable in `src/bloom_search`, the libraries for the Bloom filter and the Bloom filter tree in `src/lib/bloomfilter` and `src/lib/tree` respectively, and the ruby bindings in `src/binding`. Last but not least, unit tests can be found in the directory `src/test`. As the main executable does nothing else than parsing command line parameters and calling the libraries, we focus on those in the following sections.

Implementation of the Bloom filter

Internally, a Bloom filter is represented as an array of 64-bit integers. Each entry in the Bloom filter is then represented as a bit in one of the integers, so that a Bloom filter of size n can be represented as an array of size $\lceil \frac{n}{64} \rceil$. The functions bf_getpos() as well as bf_setpos() are used to access the individual entries without direct byte-arithmetrics. Further, functions for creating a Bloom filter from a string (bf_hash()) and obfuscation (bf_obfuscate()) are included in this module.

Implementation of the Bloom filter tree

A Boom filter tree is an opaque struct that can be created by indexing a chromosome file (tree_index_file()). The struct contains a reference to the root node, the filename, and a list of matches which will be filled during the search. Indexing the database from a file has been parallelized using the pthreads library. The algorithms for indexing and searching have been discussed in detail in Sections 2.5.2 and 2.5.3.

2.8.2 Web Service

The Bloom filter search library is written in C in consideration of speed and portability. In order to provide integration with various different infrastructures, we provide a wrapper around the library that exposes the search functionality through a Web service. The service understands *Representational State Transfer* (REST).

REST interface

Using REST, the Web service interface is exposed through HTTP verbs (GET, POST, PUT, PATCH, DELETE) that connect to a URI. The Web service is written in Ruby (as a rack-application) and can be found in the path `ws/rest-server/rest.rb`. The script expects a Ruby-C-extension built in `build/bftree.so` (which can be satisfied by using `build/` as the build directory) as well as the chromosomes downloaded in `chromosomes/`.

For the search service, only one endpoint is required: `GET /search`. The following parameters must be specified in the request:

`query`: The obfuscated Bloom filter, encoded as 0's and 1's.

`obfuscations`: The obfuscation parameter.

The service then replies with a JSON-encoded hash containing the query, the obfuscations, the time the search took, and an array of matches.

Example The client sends the following message to the server:

```
http://localhost:1338/search?
obfuscations=10&
query=00000000000001001001010001000100001000010000000101000000001011
 0000100000100000000000000000000000000100000000000000000001000001 00
```

The server responds with the following message:

```
{
  "query": "00000000000001001001010001000100000...",
  "time": 0.0500000007450581,
  "matches": [
    { "match": "AAGCT", "position": 7 },
    { "match": "ggctg", "position": 428050 },
    { "match": "GTTGC", "position": 54161548 }
  ]
}
```

Clients

Amongst others, the obfuscation of the Bloom filter happens on the client side. Since the Bloom filters are used for the actual search, it is crucial that client and server construct the Bloom filters in the same way. In the following pseudo code, *m* is the size of the Bloom filter and *k* the number of hash functions. The hash functions are all derived from a SHA hash of the query.

```
function construct_bf($str$):
    bf $\leftarrow \{0\}_{i=0}^{m-1}$
    $d \leftarrow$ bytes(SHA1($str$))
    for i in [k] {
        $\mathbf{bf}_{d_i \bmod m} \leftarrow 1$
    }
    return true
```

For writing a new client, one could either use the `bf_hash()` function in the `libbloomfilter` library or use the client in `ws/client/`. The client is written in JavaScript and runs entirely in the browser. Communication with the REST Web service is done through asynchronous HTTP requests (AJAX).

2.8.3 Asymptotic Runtime and Communication Complexity

Communication Complexity

Looking at the protocol introduced in Section 2.5.1, the only information sent from $\mathcal{U}$ to $\mathcal{S}$ is the tuple $q = (\mathbf{b}, \lambda)$. Splitting up the tuple into it's components, we get

- $|\mathbf{b}| = |\mathcal{B}^{k,m}|$ is in $\mathcal{O}(m)$, as $\mathbf{b}$ is a vector of length m,
- $|\lambda|$ is in $\mathcal{O}(\log \lambda)$, as λ has $\log \lambda$ bits in binary representation,

Because the encrypted query is just a concatenation of the components, $|q| \in \mathcal{O}(m + \log \lambda)$. The information sent from $\mathcal{S}$ to $\mathcal{U}$ is the set of results R with the size $|R| = \binom{k+\lambda}{\lambda}$. Combining the total traffic, the overall combined complexity is in

$$\mathcal{O}\left(m + \log \lambda + \binom{k+\lambda}{\lambda}\right) \approx \mathcal{O}(|s|)\,, \tag{2.5}$$

because the other parameters do not depend on the input of $\mathcal{U}$ but instead are parameters of the search scheme.

Note that the communication complexity depends only on parameters of the search protocol and not on the size of the database.

Runtime Complexity

The Bloom filter tree search resembles a binary search with runtime complexity in $\mathcal{O}(\log |A|)$ for a database A, as shown in the pseudo code for searchTree(). For each step of the traversal of the Bloom filter tree, the set membership $\in_\lambda$ has to be computed. This can be done in $\mathcal{O}(m)$ (for Bloom filter size m) using the following algorithm:

```
function included_in($\mathbf{b}$, $\mathbf{B}$, $\lambda$):
    for i in [m] {
        if ($\mathbf{b}_i > \mathbf{B}_i$) {                    // Check for mismatch
            $\lambda \leftarrow \lambda - (\mathbf{b}_i - \mathbf{B}_i)$        // use $\lambda$ to make $\mathbf{B}$ larger
            if ($\lambda < 0$) {               // ...until no tolerance is left
                return false                // ...which concludes $\notin$
            }
        }
    }
    return true
```

This concludes that the runtime complexity of the Bloom filter search is in $\mathcal{O}(m \cdot \log |A|)$.

Next, we look at the complexity of the hybrid homomorphic part. The computational

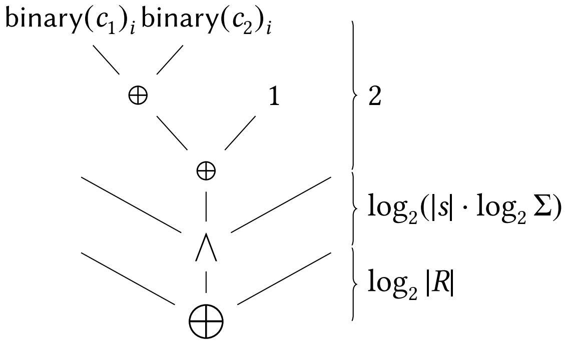


Figure 2.7: Sketch of the search circuit

complexity of this part relates to the number of gates of the circuit, which can be split into two parts as follows.

1. *Construction of the indication vector* $\mathfrak{ind}$.

   This circuit has been constructed in Equation (2.1). The inner term $(\mathfrak{b}_{ij} \oplus \mathfrak{s}_j \oplus 1)$ is conjugated over the binary representation of s and $|\mathsf{binary}(s)| = |s| \cdot \log_2 |\Sigma|$. The conjunction can be expressed as a tree of binary AND-gates with $s_1 = 2+(|s|\cdot\log_2 |\Sigma|)$ total gates.

2. *Construction of the final set of results* $\mathfrak{res}$.

   In Equation (2.4) the marked results are translated into the final set of results. The XOR over all intermediate results R can again be written as a tree of binary XOR-gates, resulting in $s_2 = |R|$ total gates.

In Figure 2.7 a schematic view of the depth of the circuit is shown. The final number of gates of the circuit is then given by

$$s_1 + s_2 = 2 + (|s| \cdot \log_2 |\Sigma|) + |R| \in \mathcal{O}(|s| + |R|) \,, \tag{2.6}$$

again because the other parameters do not depend on the inputs of $\mathcal{U}$ or $\mathcal{S}$ but instead are parameters of the search scheme.

The size of R can be controlled by the parameters of the Bloom filter k and m as well as the security parameter λ.

Putting together the Bloom filter search and the homomorphic circuits, the total runtime complexity is in

$$\mathcal{O}(\log |A| + |s| + |R|) \tag{2.7}$$

when just considering the inputs from $\mathcal{U}$ and $\mathcal{S}$.

2.9 Use Case

Moving from genetic to genomic research as well as the advances in the proteome research have resulted in sophisticated and large databases infrastructures, e.g. Ensembl (`http://www.ensembl.org/`), European Nucleotide Archive (ENA, `http://www.ebi.ac.uk/ena/`), or UniProt (`http://www.uniprot.org/`). Among different applications, these databases are used to perform sequence alignment or sequences search. In the sense of moving towards personalized medicine as a strategic future healthcare paradigm, as well as pharmacogenetics and/or pharmacogenomics as a part of new drugs development process, performing an exact search of a specific nucleotide subsequence of a patient or subject in one or more of these genomic databases is essential. A key question regarding such search is the data protection and privacy constraints of the queried sequence. For instance, genetic/genomic sequences carry indication to different phenotypes. As not all genotypes-phenotypes correlations are known to us, outsourcing any database search that involve sending subject's genomics/genetic data outside the research institute becomes risky and violate the subject's privacy in many cases. This depends on the context of the subject consent, the country to where the data is outsourced and different other aspects. All of this makes outsourcing of plain text sequence search difficult, and often forbidden due to privacy laws.

The hybrid homomorphic search introduced in this work in cooperation with the bioinformatics lab of the Leiden University Medical Center allows us to operate outsourced searches in this environment without endangering patient privacy.

2.9.1 Example

The following example should illustrate the application of the stream search to find a subsequence in a larger DNA sequence. For this purpose, let the large DNA sequence D with $|D| = 48$ be given by

$$D = \mathrm{aggtcaagtccggaatacgtacgaacgtggcagctactcgagatccga} \tag{2.8}$$

and the search term s with $|s| = 8$ given by

$$s = \mathrm{cgaacgtg}. \tag{2.9}$$

Next, choose $Q = 4 \leq \lfloor |s|/2 \rfloor$ and split D into twelve 4-grams

$$\begin{aligned} D_0 &= \text{aggt} & D_6 &= \text{acgt} \\ D_1 &= \text{caag} & D_7 &= \text{ggca} \\ D_2 &= \text{tccg} & D_8 &= \text{gcta} \\ D_3 &= \text{gaat} & D_9 &= \text{ctcg} \\ D_4 &= \text{acgt} & D_{10} &= \text{agat} \\ D_5 &= \text{acga} & D_{11} &= \text{ccga} \end{aligned}$$

and build up the Bloom filter tree from these $\mathcal{B}^{k,m}(D_i)$'s. This completes the indexing phase on the server side.

From the query, generate five 4-grams

$$\begin{aligned} s_0 &= \text{cgaa} & s_3 &= \text{acgt} \\ s_1 &= \text{gaac} & s_4 &= \text{cgtg} \\ s_2 &= \text{aacg} \end{aligned}$$

and build one query Bloom filter $\mathcal{B}^{k,m}(\{s_i\})$. Note that this query has the same property as an obfuscated query (see Section 2.4) with an obfuscation parameter of $\lambda = 16$. The complete query q consists of $(\mathcal{B}^{k,m}(\{s_i\}), \lambda, \mathfrak{s}, pk)$ where $\mathfrak{s} = \mathsf{Encrypt}(s)$ is the encrypted query. This completes the preparation phase on the client side.

The actual search is conducted using the Bloom filter search described above, using the set membership $\in_\lambda$. In this example, the set of results R will include $\{(\text{acgt}, 4), (\text{acgt}, 6), (\text{acgt}, 9)\}$. This set of results is transfered back to the client, who then checks each element of the results for an actual match.

2.10 Performance Evaluation

As an example of real world problem sizes the procedure described above was used to index and search two human chromosomes of different sizes (available at `ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes/` or using the script in the `chromosomes/` folder of the source code[1]). The experiments were run on a 2.8 GHz Intel Core i7 with 8 GB RAM. The index process utilizes all four hardware threads, the search runs only single-threaded. All timings are averages over 100 runs and can be reproduced by executing `script/benchmark.rb` in the source code package. The script

[1] `https://github.com/hperl/bf-search`

expects that the Ruby library has been build in `build/release`.

Table 2.1 and Figure 2.8 show the results for different human chromosomes. We chose Bloom filters of size 123, as 123 bits fit well into two native 64-bit integers with 5 bits left for internal flags, and 10 SHA-based hash functions to have enough room left for obfuscation as well as set-based Bloom filters. The percentage corresponds to the ratio of $\frac{\text{real results}}{\text{noise}}$. For example, hiding in 5 % means that only 5 % of the results are real, the other 95 % of the results was noise added because of the Obfuscated Bloom filters. The baseline performance is given as a search with no obfuscation added.

It should be noted that using our approach the database containing the Chromosomes only needs to be indexed once independently of the number of users querying the database.

			Search (hiding in x %)			
	File size	**Index**	Baseline	11 %	5 %	1 %
chr1.fa	254 MB	3.62 s	0.016 s	0.056 s	0.078 s	0.096 s
chr2.fa	248 MB	3.56 s	0.011 s	0.046 s	0.079 s	0.096 s
chr3.fa	201 MB	2.74 s	0.009 s	0.038 s	0.063 s	0.076 s
chr4.fa	194 MB	2.65 s	0.008 s	0.036 s	0.061 s	0.073 s
chr5.fa	184 MB	2.50 s	0.008 s	0.034 s	0.056 s	0.070 s
chr6.fa	174 MB	2.49 s	0.008 s	0.032 s	0.060 s	0.071 s
chr7.fa	162 MB	2.24 s	0.007 s	0.030 s	0.049 s	0.061 s
chr8.fa	149 MB	2.04 s	0.007 s	0.029 s	0.047 s	0.057 s
chr9.fa	144 MB	2.10 s	0.005 s	0.023 s	0.039 s	0.047 s
chr10.fa	138 MB	1.84 s	0.006 s	0.025 s	0.042 s	0.051 s
chr11.fa	137 MB	1.82 s	0.006 s	0.025 s	0.042 s	0.051 s
chr13.fa	117 MB	1.59 s	0.004 s	0.018 s	0.031 s	0.037 s
chr14.fa	109 MB	1.48 s	0.004 s	0.016 s	0.028 s	0.034 s
chr15.fa	104 MB	1.38 s	0.004 s	0.015 s	0.026 s	0.032 s
chr16.fa	92 MB	1.22 s	0.004 s	0.015 s	0.025 s	0.031 s
chr17.fa	82 MB	1.11 s	0.003 s	0.014 s	0.024 s	0.030 s
chr18.fa	79 MB	1.04 s	0.003 s	0.014 s	0.024 s	0.030 s
chr19.fa	60 MB	0.79 s	0.002 s	0.010 s	0.018 s	0.022 s
chr20.fa	64 MB	0.85 s	0.003 s	0.011 s	0.019 s	0.023 s
chr21.fa	49 MB	0.67 s	0.001 s	0.007 s	0.012 s	0.015 s
chr22.fa	52 MB	0.72 s	0.001 s	0.006 s	0.011 s	0.014 s

Table 2.1: Performance of the Obfuscated Search

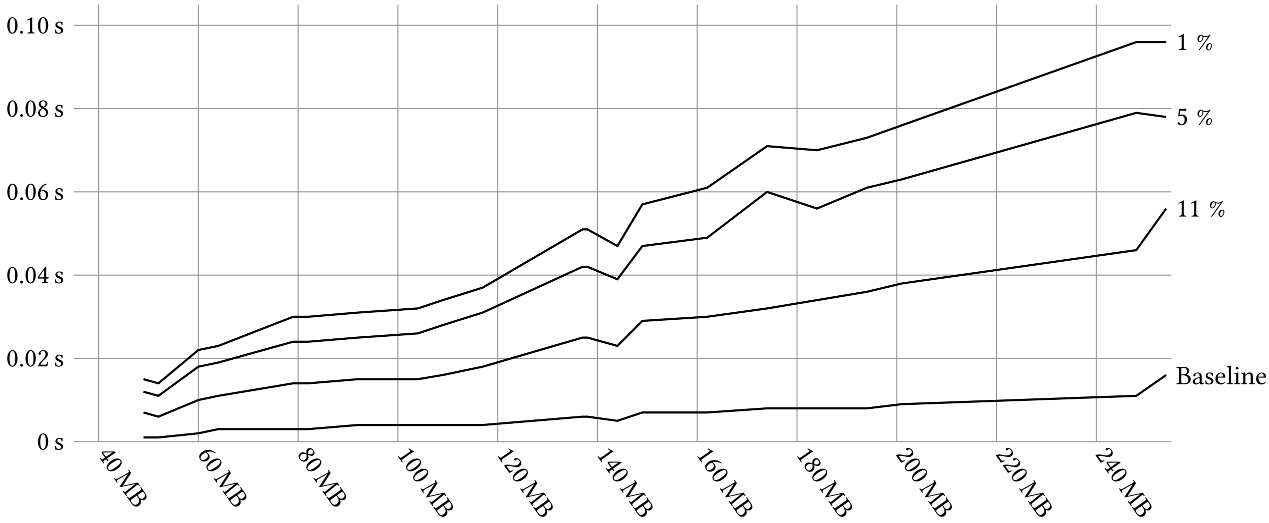


Figure 2.8: Plot of the Performance of the Obfuscated Search

As stated above, these results can be processed further using hCrypt due to the small size of the results generated by the Bloom filter search. The performance figures of an exact match search using hCrypt for different set of results and search term sizes are outlined in Table 2.2.

	Size of the set	
Searchterm size	100 elements	1000 elements
5 Bit	0.007 s	0.067 s
16 Bit	11.300 s	115.000 s
32 Bit	27.800 s	288.000 s
64 Bit	55.600 s	560.000 s

Table 2.2: Performance of homomorphic post-processing.

2.10.1 Performance Comparison with PIR Schemes

We will use data from Costea et al. [27] for a performance comparison between the Obfuscated Bloom filter search and Private Information Retrieval (PIR) techniques. Costea et al. have implemented two PIR schemes: One based on the *Quadratic Residuosity Assumption* (QRA-PIR) by Kushilevitz and Ostrovsky [12], and one based on the ϕ*-hiding*

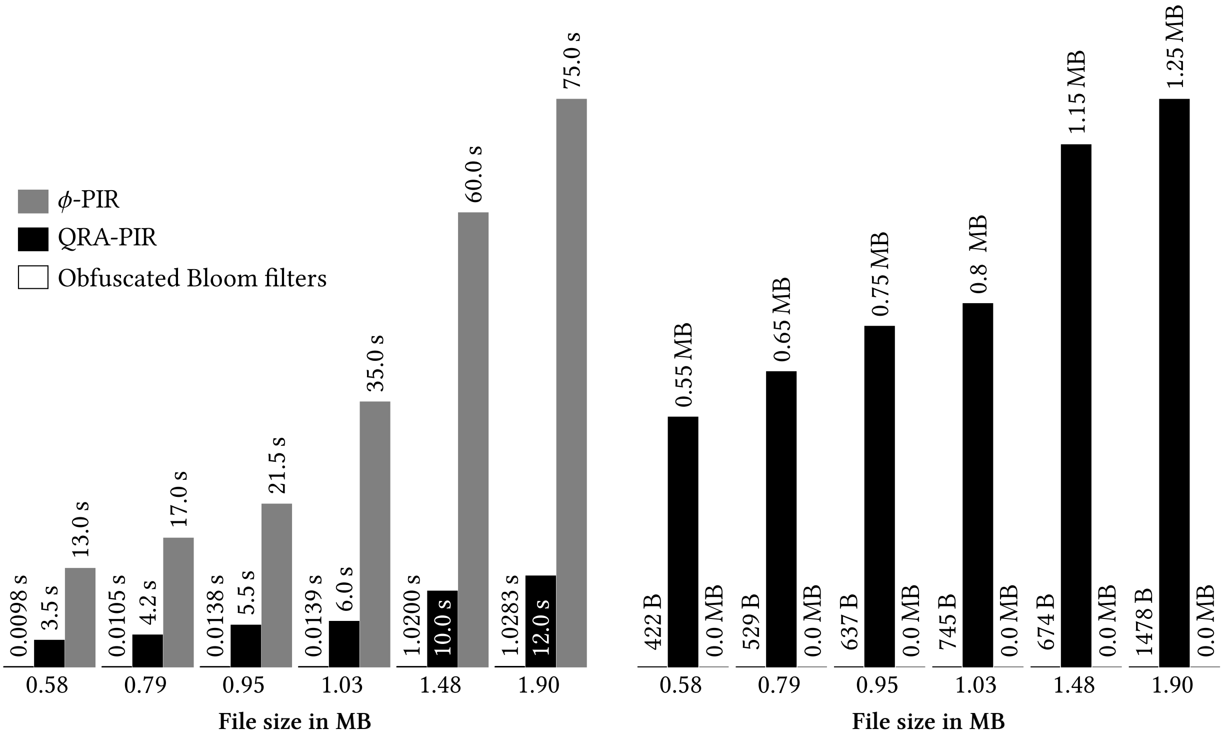
Assumption (ϕ-PIR) by Gentry and Ramzan [11]. Both implementations were written in C++, which is roughly comparable to the C implementation of the Bloom filter search.

The PIR implementations operate on location-based data, but the general search scenario is the same: A private search for a point of interest (POI) in a database. We assume that a POI has a size of 32 bytes, i.e. it just contains the GPS coordinates. Figure 2.9 shows the performance comparison of the two PIR schemes along with the performance of the Obfuscated Bloom filter search. Because the differences between the compared systems is so huge, the numbers can also be seen in Table 2.3. Since Costea et al. did not publish their source code for the PIR schemes, we relied on their measurements, which we took from the graphs in [27, Fig. 3–4]. For the communication size of ϕ-PIR, the graphs did not show a significant deviation from zero, which is why we set the column to 0.0 MB. Still, at least a single result would need to be transmitted.entirely on the server, we do not consider the client CPU timings of [27].

For the server CPU timings, OBF is significantly faster than both PIR schemes, even when hiding in 1 % of the results. Search times of 12 s for QRA-PIR and even 75 s for ϕ-PIR for a file under 2 MB size shows that even recent PIR schemes are still far from achieving the performance needed to search in large datasets common in biology. Considering the communication size, OBF is about as good as ϕ-PIR, and a drastic improvement over QRA-PIR (OBF: 1478 B, QRA-PIR: 1.25 MB). When combining the server CPU time and the communication size, the compared PIR schemes are either fast on the server or light on the wire, but not both. However, PIR still promises a stronger security, yet without the possible trade-off to sacrifice a little bit of security for a large performance boost.

	Server CPU (time)			**Communication** (size)		
File size	QRA	ϕ-PIR	OBF	QRA	ϕ-PIR	OBF
0.58 MB	3.5 s	13.0 s	0.0098 s	0.55 MB	0.0 MB	422 B
0.79 MB	4.2 s	17.0 s	0.0105 s	0.65 MB	0.0 MB	529 B
0.95 MB	5.5 s	21.5 s	0.0138 s	0.75 MB	0.0 MB	637 B
1.03 MB	6.0 s	35.0 s	0.0139 s	0.80 MB	0.0 MB	745 B
1.48 MB	10.0 s	60.0 s	0.0200 s	1.15 MB	0.0 MB	674 B
1.90 MB	12.0 s	75.0 s	0.0283 s	1.25 MB	0.0 MB	1478 B

Table 2.3: Comparison of the Bloom filter search with two different PIR schemes



(a) Comparison of Server CPU time

(b) Comparison of Communication size

Figure 2.9: Comparison of the Bloom filter search with two different PIR schemes

2.11 Conclusion

In this work, we introduced a search algorithm that utilizes Obfuscated Bloom filters to ensure the confidentiality of search queries as well as the results of the search. Through obfuscation, the query can be made secure enough to conform to data protection regulations, yet the set of results is small enough to allow further processing with hCrypt, e.g. an exact-match search as demonstrated in this work. Our approach achieves a communication complexity of $\mathcal{O}(|s|)$ as well as the runtime complexity of $\mathcal{O}(\log |A| + |s| + |R|)$ with a flexible parameter to adjust the size of the set of results R.

Also, the size of the database that can be searched is sufficiently large and the search itself is fast enough ($\mathcal{O}(\log |A|)$) for first real world use cases to be practically implemented. We showed extensive performance analysis and comparisons with PIR schemes. We presented a security analysis of our design and demonstrated its feasibility using datasets containing human chromosomes. This is one of the first systems to enable the practical use of hCrypt. Beyond offering a solution for search with encrypted terms, this work also serves as an example of how systems can be designed to incorporate the new possibilities of Homomorphic Encryption.

CHAPTER 3

Evaluation of SSL Validation Systems

Although SSL/TLS is in widespread use today, certificate validation currently suffers from the weakest link property created by the fact that any trusted CA can sign a certificate for any domain. Thus, if a single CA is compromised or coerced, any and all hosts using CA-signed certificates can be endangered. Over the years, there have been many proposals on how to fix this problem, but only one is en route to real world deployment right now: Certificate Transparency.

In this chapter we look at how Certificate Transparency and other approaches aim to strengthen the security of SSL validation and evaluate their respective deployability properties. We conduct a thorough evaluation of the current CA-based PKI and the alternative approaches Perspectives, Convergence, Certificate Transparency, Sovereign Keys, TACK and DANE. Finally, we have identified a list of open problems on the road to a better TLS public key infrastructure.

The work in this chapter was presented as a poster at the 2013 IEEE Security & Privacy conference, where Clark and Oorschot also published their SoK on SSL and HTTPS [28]. This was joint work with Sascha Fahl, who helped me refine the description of the benefits and with whom I also collaborated on other research concerning the SSL landscape [29–31]; as well as Matthew Smith, who did general counseling. The rest is my own work.

3.1 Introduction

The TLS/SSL protocol is one of the mainstays of Internet security. However, unrest is growing as more large scale compromises and real world MITMAs are discovered. This reflects the fact that the current certificate authority based public key infrastructure (CA-PKI) is a prominent example of a weakest-link security system: Since all trusted root

CAs can issue certificates for any domain, an attacker can pick the weakest or most coercible CA to target for an attack — and a single vulnerable, malicious or coercible CAs undermines the security of the entire system. To make matters worse, these attacks can go unnoticed quite easily. According to the EFF's SSL Observatory, current browsers trust roughly 1500 different CAs from roughly 650 different organizations [32].

As an example, the following security breaches have recently made the news:

- In 2010, VeriSign was compromised, allowing the attackers to issue arbitrary certificates [33].
- In March 2011, an attacker from Iran was able to compromise the Comodo certificate authority [34] and get certificates for www.google.com, login.yahoo.com, login.skype.com, addons.mozilla.org, and login.live.com. A MITMA attack with at least one these certificate was observed.
- In August 2011, attackers used the DigiNotar CA to issue at least 200 fraudulent certificates and used them to impersonate web servers [35]. The breach eventually lead to the exclusion of the CA from most browsers and operating systems.

All these incidents were discovered after the fact by vigilant users and security researchers, who detected the man-in-the-middle attacks (e.g. Google pinning its own certificates in Chrome). The CAs were either unaware or hiding the fact that they had been compromised. Either way this set of circumstances is highly critical, since most sites do not use certificate pinning and most users are not capable of detecting a MITMA with a valid certificate.

This shows that the flaws in the current system are already being exploited by attackers. It should also be considered that several government agencies legally own trusted root CAs and thus can also execute MITMA attacks against arbitrary users and sites. While it is only speculative whether these kind of organizations actually use their CAs to mount MITMAs, it should be considered that they have the power to do so without much effort. Rogue states are also a concern in this scenario.

Due to these obvious issues with the current SSL validation mechanism, a number of proposals have been made that promise to strengthen the security of certificate validation. Some of the most prominent approaches are: Perspectives [36], Convergence [37], Certificate Transparency [38], Sovereign Keys [39], TACK [40], and DANE [41]. Of these, only Certificate Transparency is currently being rolled out, i.e. used for Extended Validation (EV) certificates since January 2015.

This wealth of new solutions also inspired Clark and Oorschot [28] to publish a Systematization of Knowledge (SoK) paper on certificate trust model enhancements in 2013. Instead of evaluating the proposed solutions directly, the authors evaluated primitives that were used by those proposals. Certificate Transparency is only evaluated under the primitive “List of Active Certificates”, which according to the authors offers only the property of “Responsive Revocation”. The authors further assert CT the primitive “Multipath Probing”. However, although multipath probing could be used with CT, it is deliberately not included in the proposal, as side-channels during the handshake can cause failures, e.g. when connecting to a captive portal that blocks all other internet access. The CT team found that those extra connections fail at least 1 % of the time[1].

We are now in the comfortable situation that we can look at both the other proposed solutions and see what they are missing in comparison to Certificate Transparency, as well as what [28] is missing in the evaluation of Certificate Transparency. We proceed by carefully revisiting both the SoK as well as CT and extract the essence of the proposal that make CT feasible for real world deployment. We then take another look at the proposals with a focus on the properties that helped CT be deployable. We introduce an evaluation framework based on a catalog of desirable benefits of SSL validation systems. We identify the different strengths and weaknesses of the systems, try to shed light on the trade-offs all systems have to make and point out which disadvantages they incur that currently hinder adoption.

The contribution of our work is as follows:

- We propose an evaluation framework to compare and discuss SSL certificate validation schemes to help organize and formalize the ongoing debate in this area.
- We evaluate Perspectives, Convergence, Certificate Transparency, TACK, Sovereign Keys, AKI, ARPKI and DANE using our framework and highlight the comparative strengths and weaknesses.
- We identify a list of open problems on the road to a better TLS public key infrastructure.

3.2 Related Work

Clark and Oorschot [28] published a systematization of knowledge paper on SSL and HTTPs where they revisited issues of the crypto protocol as well as the trust model between the CAs and the browsers. Then, they introduce an evaluation system for primitives that improve the CA/browser trust model. The primitives were split into three groups as follows.

In the first group, key pinning (as in TACK or TOFU), multipath probing (as in Perspectives, Convergence), and key agility (as in AKI) are all primitives for detecting man-in-the-middle attacks. This is also the main focus of our evaluation system work.

[1] http://www.certificate-transparency.org/comparison

The second group contains primitives that are concerned with SSL stripping protection. Here, the authors evaluate HTTPS-only pinning on the server, the client (through a preloaded list), and through DNS.

The third group of primitives is related to revocation. They evaluated CRLs, OCSP-stapling, short-lived certificates, and a list of active certificates, like the certificate log in CT.

However, the author's systematization fails to show how and if the combination of primitives can inherit the benefits of both primitives. In the next section, we will focus intensively on CT, since it is the only system with an active deployment plan. From looking at just the evaluation in [28], it is puzzling that CT can be thought of as a system improving the security of TLS, since it was only evaluated for revocation.

3.3 Certificate Transparency

Certificate Transparency [38] (CT) is currently being developed by Ben Laurie, Adam Langley and Emilia Kasper at Google. When a certificate for a server has been signed by a CA, the server or the CA submits the certificate to a log server. There need to be a small number of log servers that periodically check each other for consistency as well as query updates. The log servers insert the certificates into append-only data structures based on Merkle hash trees. The hash trees ensure that tampering with the log is easily detectable and would thus revoke trust in the log server. If a new certificate is added to the log, a new branch with a new root will be inserted into the tree, the old root eventually becoming a child of the new root. This means that the new tree can still prove consistency for an older log.

CT was designed specifically for real-world requirement of no side-channels. During a TLS handshake, there should be no other connections necessary. This is important, since those extra connections might fail, or even just slow down the handshake significantly. We modeled this requirement in D12 *(Quasi-) No-Out-Of-Band-Connection.*

This is realized in CT through embedding a signed certificate timestamp (SCT) assuring that the certificate was logged either into the certificate (delivered through an X.509 extension) or in an OCSP response (delivered through OCSP stapling, a TLS handshake extension). The biggest advantage of CT however is, that not all clients even need to check these extensions. The main problem CT solves is that of CAs issuing fraudulent certificates unnoticed. By collecting all issued certificates, CT enables inspection of which CA issued which certificates. As long as some percentage of the clients check the SCTs, certificates without SCTs will get reported. And as long as some CAs as well as domain owners check that no other CA issued a certificate without explicit order from the domain owner, those malicious CAs will be detected. Misbehaving CAs can then be named and blamed, and the fraudulent certificates can be revoked.

There is a high incentive for CAs to be perceived as reliable, since ultimately browser and operating system vendors decide which CA certificates they include in their trust stores, and a CA's signature is only valuable as long as it is included into all major trust stores. For example, after the DigiNotar hack, vendors decided to no longer include DigiNotar's root certificate.

So while CT does not prevent man-in-the-middle attacks from happening right after a CA has been hacked, it does strengthen the whole CA ecosystem by detecting publicising compromises very quickly.

For an in-depth discussion of CTs and other validation systems' advantages and disadvantages we introduce a list of benefits in the next section and then evaluate the Systems in Sections 3.5–3.6.

3.4 An Evaluation Framework for SSL Validation

In this chapter we suggest such a framework loosely following the design of the framework presented by Bonneau et al. [42] for evaluating web authentication schemes. Our findings are filed into one of two categories: deployability, or security. In each category, we describe a set of benefits, consisting of a mnemonic title and a description. In addition, we indicate the correlation to the benefit from [42] when applicable.

3.4.1 Deployability Benefits

In this category deployment aspects are evaluated.

D 1 *No-User-Cost:* The cost for the user of the scheme is negligible. This means for instance that the system does not require special hardware which would need to be purchased by the end user. (c.f. [42, D2: *Negligible-Cost-per-User*]).

D 2 *No-Server-Cost:* The total cost per user of the scheme is negligible for the server. As opposed to D 1 we only consider costs for the server here. We say that additional CPU or bandwidth costs are negligible, but additional recurring fees are not. A system has *Quasi-No-Server-Cost* if it is possible to enrol a server without costs but it is common to use a paid service. (c.f. [42, D2: *Negligible-Cost-per-User*]).

D 3 *Server-Compatible:* On the server side, the system is compatible with SSL and X.509 certificates. Servers don't need to patch their web server or SSL library. (c.f. [42, D3: *Server-Compatible*])

D 4 *Browser-Compatible:* On the browser side, the system is compatible with SSL and X.509 certificates. The browser doesn't need to be patched. (c.f. [42, D4: *Browser-Compatible*]).

D 5 *Incrementally-Deployable:* The system can be deployed incrementally. The full benefit should only be awarded if early adopters already benefit from the system even if wide-spread adoption has not occurred yet. The benefit should not be granted if the benefit to adopters only kicks in once everybody has migrated, even if the technical migration can be executed in an incremental and backwards compatible way. A system is *Quasi-Incrementally-Deployable* if early adoption is beneficial and safe for the case that each client can securely acquire a list of servers that already offer the new service, i.e. no downgrade attacks are possible.

D 6 *Negligible-Communication-Overhead:* The total communication overhead between the client, the server, and (potentially) a third party is negligible. The system is plausible for mobile devices and settings with a low bandwidth connection.

D 7 *Negligible-Computational-Overhead:* The total computational overhead combined for the client, the server, and (in some cases) a third party is negligible. The system is plausible for mobile devices and settings with low processing power. The system has *Quasi-Negligible-Computational-Overhead*, if the computational overhead is negligible for the user.

D 8 *No-Additional-Infrastructure:* For the deployment, no additional infrastructure is necessary. The system either reuses the CA or DNS infrastructure or doesn't need any at all. We award a *Quasi-No-Additional-Infrastructure* if the systems reuses the CA or DNS infrastructure but requires those services to integrate the system.

D 9 *Trusted-Root-CA-support:* The system can validate trusted root-CAs that are distributed out-of-band, e.g. in the Browser or OS.

D 10 *Custom-Root-CA-support:* The system can integrate custom root-CAs that are only trusted by clients that explicitly choose to trust them. This enables organizations to use the system to develop a closed PKI-infrastructure that can be combined with the public structure. A system has *Quasi-Custom-Root-CA-support* if user action is required to include custom root-CAs.

D 11 *Selfsigned-Certificate-support:* The system can validate certificates that are not signed by any third party.

D 12 *No-Out-Of-Band-Connection:* For the client to verify the server, the client only needs connectivity to the server, not to any other third party. We award *Quasi-No-Out-Of-Band-Connection*, if an out-of-band connection is only required in rare cases, e.g. if the certificate has just been issued or to download periodic updates.

D 13 *X.509-compatible:* The certificates used for authentication are compatible with X.509. For example, systems that require or support multiple CA signatures in one certificate are not X.509-compatible, since the X.509 structure only allows for one issuer for a certificate.

3.4.2 Security and Privacy Benefits

S 1 *Built-In-Revocation*: The system has build in capability to revoke certificates. This can become necessary in the case when a server's private key was stolen. We allow for a short grace period bound by the network delay, before the compromised credentials must be revoked. The full benefit should only be awarded if the revocation mechanism is built into the system as an integral component.

S 2 *OCSP-or-CRL-Compatibility*: The system can support revocation through OCSP or CRL.

S 3 *Resilient-To-DOS-Attacks*: The system does not rely on an infrastructure that is required for validation that can be knocked out through a denial of service. It is *Quasi-Resilient-To-DOS-Attacks* if an out-of-band connection is only needed from time to time or in special cases. This benefit is closely linked to 8.

S 4 *User-Privacy-Preserving:* When using the system for server authentication, the information that the client is visiting a specific server does not leak to any third party.

S 5 *Secure-Key-Migration:* When a domain's owner changes keys this can be done in an automatic and verifiable way. The system has *Quasi-Secure-Key-Migration* if migration is possible but for the client the process is indistinguishable from a MITMA.

S 6 *Secure-Key-Migration-After-Credential-Theft:* When a domain owner changes keys because a credential theft was detected, this can be done in an automatic and verified way. The system has *Quasi-Secure-Key-Migration-After-Credential-Theft* if migration is possible but the process is indistinguishable from a MITMA for the client.

S 7 *Secure-Domain-Migration*: The system allows the owner of a domain to change in a verifiable way. This benefit may not be awarded if this process leaves the previous owner the capability to impersonate the new owner for any amount of time. A system has *Quasi-Secure-Domain-Migration* if migration is possible but for the client the process is indistinguishable from a MITMA.

S 8 *First-Contact-Protection*: The system protects the connection fully from the very first connection from the client. The benefit should not be award if it is a "trust on first use" system. The system offers *Quasi-First-Contact-Protection* against an adversary if an incident is detectable after the fact. This benefit is evaluated according to the adversary capabilities in Section 3.4.2.

S 9 *Connection-Protection:* The system protects the adversary from eavesdropping on the connection. This is equivalent to server impersonation. The system offers *Quasi-Connection-Protection* against an adversary if an incident is detectable after the fact. This benefit is evaluated according to the adversary capabilities in Section 3.4.2.

Adversary Capabilities

For a more precise security analysis of benefits S 8 and S 9 we consider adversaries of different capability levels.

Level 1. Active MITMA required The adversary only controls the connection between the client and the server.

Level 2. Trusted CA certificate required Additionally, the adversary can sign any certificate using an arbitrary trusted root-CA (i.e. a "weakest link" attack).

Level 3. Compromising user chosen third parties required Additionally, the adversary can compromise any *n* third parties of his choice (i.e. *n* "strongest links" attack).

Defense against a first-level adversary is already covered by the current CA-PKI. The minimum improvement needed by any system is to protect against an attacker at level 2. Optionally a system also requires the attacker to successfully compromise or knock out *n* further third parties. Ideally this is combined with *trust agility*, meaning that the user can choose which *n* parties are used to validate the connection thus making it even harder for the attacker since he either has to know which *n* parties were chosen or compromise all possible third parties.

3.4.3 On Usability

We decided not to include usability benefits in the catalog for specific reasons. Even though the user experience of a SSL validation system may have a large impact on the security of the system e.g. through the way warning messages as well as safe connections are represented, we found it impossible to judge the various merits of the different systems objectively. While there might be differences in how helpful a system is when compiling information for a warning message, ideally a system should not show any false-positive warnings at all. Yet, no SSL system purposely introduces the possibility for false positives. When they occur, they occur because the server was misconfigured.

Since all can be theoretically be misconfigured and both this and the quality of the warning messages are influenced more by implementation and deployment than the system itself, we decided not to score these aspects. In many cases the user interface will look the same as it looks today, with any other SSL system under the hood. Writing good warning messages is certainly a big challenge, with a whole bag of definitions on what "good" means in this context and should be evaluated separately.

The ambiguousness of the warning message depends on the number of false positives. If false positives are more probable, warning messages have to account for the possibility that everything could be OK. So in that sense, the usability of a SSL system is directly related to how easy it is to set up for a server administrator. We cover this as part of the deployability benefits. Finally, the debate of whether validation systems should block the connection if in doubt has been discussed by Sunshine et al. [3] and applies to all proposals as well.

3.5 Evaluation of SSL With A CA-PKI

We use the current SSL system based on a Certificate Authority Infrastructure (CA-PKI) as a baseline and to make the reader familiar with the evaluation framework.

Server authentication with a CA-PKI is made up of the following steps: Prior to the connection attempt by the client, the server has obtained a signature for its certificate from a trusted certificate authority. The certificate authority has made sure that the person issuing the certificate request has access to the specified domain, then signed the certificate. The certificate is sent to the client as part of the SSL handshake. The client checks the certificate for a signature by a trusted certificate authority, and on success concludes that the server is authentic.

As the CA-PKI is already fully deployed, it has a head start in the deployability aspects. It has *No-User-Cost* as well as *Quasi-No-Server-Cost*. While there are trusted CAs (such as StartSSL) that offer free certificates, it is very common for server to used payed for certificates, especially for services like Extended Validation or wildcard certificates. It is *Server-Compatible*, *Browser-Compatible*, and *Incrementally-Deployable* and trivially has the *No-Additional-Infrastructure* benefit. While there are alternative systems which are server compatible, all of the current alternatives require the browser to execute non standard steps and thus currently require browser plugin to enable this. This benefit will become relevant when one or more of the systems reaches a higher level of maturity and starts getting adopted by browser vendors.

We rate the CA-PKI as having *Negligible-Communication-Overhead* and *Negligible-Computational-Overhead* and use it as a baseline for the evaluation of the alternative systems. The CA-PKI has *Trusted-Root-CA-support* and *Quasi-Custom-Root-CA-support*, since the user is required to install additional CA certificates on the client. *Selfsigned-Certificates* are not supported (the user is presented with a warning when connecting to a self-signed certificate). It needs no *Out-Of-Band-Connections*.

CA-PKI has no *Build-In-Revocation*, but is *OCSP-or-CRL-Compatible*. In order to keep the complexity manageable we do not consider knock on effects of OCSP such as the out-of-band communication in the main system, but the reader should be aware that both OCSP and CRL come with their own sets of benefits and problems. CA-PKI is however *Resilient-To-DOS-Attacks* since there is no third party involved during the connection attempt and the protocol is not asymmetrically in favor of the attacker. Considering the security benefits, the CA-PKI is *User-Privacy-Preserving* since no connections other than to the server are needed. It follows the same protocol regardless of whether a client connects for the first time or not, hence it has *First-Contact-Protection* and *Connection-Protection* against an adversary that can only do an active MITMA. In order to successfully execute a MITMA, an attacker needs to intercept the connection (*Active-MITMA-Required*) and additionally requires a valid certificate from *any* trusted root CA (*Trusted-CA-Certificate-Required*). CA-PKI does not get the benefit *Compromising user chosen third parties required*, as indicated by the empty arrow in Table 3.1, which means that a successful attack against a single CA renders the whole scheme vulnerable.

The CA-PKI can handle *Secure-Key-Migration* with or without credential loss, as a newly signed certificate is just as valid as the last one. However, *Secure-Key-Migration-After-Credential-Theft* renders the connections vulnerable against an adversary capable of an active MITMA. Note: This can of course be counter using an add-on like OCSP, however, as stated above we do not mix the benefits of the systems in this round of evaluation. The CA-PKI does not provide *Secure-Domain-Migration*: The old owner of the domain still has a valid certificate that can be used for an attack; likewise the new owner could fool domain visitors into thinking that they are still seeing the old site.

3.6 Evaluation of Alternative Validation Systems

3.6.1 Perspectives

The *Perspectives* system proposed by Wendlandt, Andersen and Perrig [36] is an infrastructure of distributed semi-trusted *notary* servers that periodically record the public keys of SSL protected servers on the web. Each notary keeps a record containing information about which server used which certificates in the past. Trusted CAs are not needed.

																									Capabilities				
Scheme	Ref.	**Deployability benefits**	No-User-Cost	No-Server-Cost	Server-Compatible	Browser-Compatible	Incrementally-Deployable	Negligible-Communication-Overhead	Negligible-Computational-Overhead	No-Additional-Infrastructure	Trusted-Root-CA-support	Custom-Root-CA-support	Selfsigned-Certificate-support	No-Out-Of-Band-Connection	X.509-Compatible	**Security benefits**	Built-In-Revocation	OCSP-or-CRL-Compatibility	Resilient-To-DOS-Attacks	User-Privacy-Preserving	Secure-Key-Migration	Secure-Key-Migration-After-Credential-Theft	Secure-Domain-Migration	Active MITMA required	Trusted CA certificate required (weakest link)	Compromising user chosen third parties required (strongest link)	First-Contact-Protection	Connection-Protection
SSL with CA-PKI	–		●	◐	●	●	●	●	●	●	●	◐	○	●	●		○	●	●	●	●	○	○	➡			●	●
(90's)																									➡		○	○
																										Σ0⇨	○	○
Perspectives	[36]		●	●	●	○	●	○	●	○	●	●	●	○	●		○	●	○	○	◐	◐	◐	➡			●	●
(2008)																									⇨		●	●
																										➡*n*➡	○	○
DANE	[41]		●	◐	●	○	○	●	●	◐	●	●	●	●	●		○	●	●	●	●	●	●	➡			●	●
(2010)																									⇨		●	●
																										➡1➡	○	○
Convergence	[37]		●	●	●	○	●	○	●	○	●	●	●	○	●		○	●	○	●	◐	◐	◐	➡			●	●
(2011)																									⇨		●	●
																										➡*n*➡	○	○
Sovereign Keys	[39]		●	◐	○	○	◐	○	●	○	●	○	○	○	●		●	●	○	○	●	●	●	➡			●	●
(2011)																									➡		●	●
																										➡1➡	◐	◐
Certificate Transparency	[38]		●	◐	●	○	◐	●	●	○	●	○	○	◐	●		○	●	◐	●	●	●	●	➡			●	●
(2012)																									➡		●	●
																										➡1➡	◐	◐
CT + Revocation	[43]		●	◐	●	○	◐	●	●	○	●	○	○	◐	●		●	●	◐	●	●	●	●	➡			●	●
Transparency																									➡		●	●
(2014)																										➡1➡	◐	◐
TACK	[40]		●	●	○	○	●	●	●	●	●	●	●	●	●		○	○	●	●	●	◐	◐	➡			○	●
(2012)																									⇨		○	●
																										Σ0⇨	○	●
AKI	[44]		●	◐	○	○	●	●	●	○	●	●	○	●	○		●	○	●	●	●	●	●	➡			●	●
(2013)																									➡		●	●
																										➡*n*➡	◐	◐
ARPKI	[45]		●	◐	○	○	●	●	●	○	●	●	○	●	○		●	○	●	●	●	●	●	➡			●	●
(2014)																									➡		●	●
																										➡*n*➡	◐	◐

● = offers the benefit; ◐ = almost offers the benefit, ○ = does not offer the benefit.

➡ = step required; ⇨ = step not required for MITMA.

Table 3.1: Evaluation of SSL validation systems

When a client wants to authenticate a server, the client queries *N* notary servers and compares their stored certificates for the server with the certificate presented by the server. If the certificate presented matches the certificates seen by the notaries then it is unlikely that there is an active MITMA (which would use a different certificate than the one observed by the notaries). To succeed the attacker would have had to successfully compromise or MITMA all *N* user selected notaries as well as the connection between the user and the server. Since the notaries report the same state to all queries in the MITMA case an attacker would have to additionally MITMA all connections to the target server and all connection to all notaries to remain undetected. The number and choice of notaries as well as how many notaries need to confirm the server's certificate can be configured by the client.

As deployability benefits, Perspectives has *No-User-Cost* as well as *No-Server-Cost.* Although servers still need certificates, it does not matter whether they are self-signed, or signed by a trusted or custom root CA (*Trusted-Root-CA-support*, *Custom-Root-CA-support*, *Selfsigned-certificate-support*). It is *Server-Compatible*, because notaries work with the standard X.509 certificates. It is however not *Browser-Compatible*, because clients need to query the notaries for verification. The system is *Incrementally-Deployable*, meaning that as soon as enough notaries are available, clients could use the system even though no one else uses it. However it does not have *Negligible-Communication-Overhead*, as connections to the notaries need to be made during the request. As Perspectives requires a separate notary infrastructure which is different from the current CA-PKI, it has no *No-Additional-Infrastructure.* Further, *Out-Of-Band-Connections* are required.

Perspectives has no *Built-In-Revocation*, as no mechanism for revocation exists and the notaries can not tell the difference between a second valid certificate and one that got created because the first certificate has been stolen. However, it has *OCSP-or-CRL-Compatibility*, and the revocation could even be handled by the notaries directly. Further, it is not *Resilient-To-DOS-Attacks*, as it requires the notary infrastructure to be online during certificate validation. It is not *User-Privacy-Preserving*, because the browser history leaks to the notaries. It has *Quasi-Secure-Key-Migration* (with or without credential theft), as the new certificate will get picked up by the notaries automatically, but this is not distinguishable from a MITMA from a client's perspective. For a MITMA, an attacker would need to compromise the answer of the *N* notaries the client chose. Against level 1 and 2 adversaries, Perspectives offers *First-Contact-Protection* and *Connection-Protection.*

3.6.2 Convergence

Convergence by Marlinspike [37] is based on Perspectives. Both systems use notaries to verify the authenticity of the server. However, in convergence, notaries can use validation strategies other than simply requesting the server's certificate via HTTPS. Possible strategies are: DNSSEC, BGP data, or "SSL observatory" results. Furthermore, a two-step onion routing protocol is used for the queries to the notaries to preserve the privacy of the client.

As Convergence and Perspectives follow a very similar design, we only focus on the difference here. Convergence is *User-Privacy-Preserving*, since the IP address of the client does not reach the notary during a query.

3.6.3 Certificate Transparency

See Section 3.3 for an introduction to Certificate Transparency.

The system has *No-User-Costs* as well as *Quasi-No-Server-Cost-per-User*, as a server need a signed X.509 certificates.

CT is not *Browser-Compatible*: The automated client-side request for appearance of a server's certificate in the append-only logs has to be integrated into the Browser for CT to work. It is not *Server-Compatible*, since servers need to pass extra information in the SSL handshake. CT is *Quasi-Incrementally-Deployable*, as a MITM could omit CT related information during the attack. However, this can be detected by the client either if the client has previously connected to the host in question or by looking up the server in a list of CT-ready hosts. While it is in theory possible to setup CT incrementally and with respect to backwards compatibility, the true benefit of basing trust decisions on the appearance of a certificate in the append-only logs is only applicable when the switch to CT is complete. CT has both *Negligible-Communication-Overhead* and *Negligible-Computational-Overhead*, as the proof of validity is included directly in the certificate. With the log servers, a new infrastructure is needed, thus CT does not get the benefit *No-Additional-Infrastructure. Trusted-Root-CA-support* is supported, however it is currently not possible to add *Custom-Root-CA-support*[2] or *Selfsigned-Certificate-support*, since certificates need to be signed by a trusted root-CA *known to the log server.* Since the client must sometimes connect to the CT server CT only gets the *Quasi-No-Out-Of-Band-Connection.*

As revocation is not part of CT, it only has *OCSP-or-CRL-Compatibility.* It is only *Quasi-Resilient-To-DOS-Attacks* since an attacker could force a situation in which the client needs an update from the CT server which the attacker could then deny. CT is *User-Privacy-Preserving*, as only the roots of the Merkle trees are sent from the CT server to the client. The CT server does not receive any information about browsing behaviour. CT has *Secure-Key-Migration* with or without credential theft through resubmission to the log. It does allow for *Secure-Domain-Migration*, because the previous owner of the

[2] There is ongoing work that could address this in the near future

domain can't attack the current owner once he submitted the new certificate to the log and it is always visible that a new key was registered. Finally, in order to MITMA a connection, the adversary would need to acquire a valid certificate as well as manipulate the log server so that it adds the certificate to the log. While the MITMA in his case would be successful it would also be detectable in the logs of the CT server. Thus we award a partial benefit even for this attack. CT offers both *First-Contact-Protection* and *Connection-Protection.*

3.6.4 CT + Revocation Transparency

CT + Revocation Transparency [43] (CTRT) is an extension to CT proposed by Mark Ryan. In addition to the chronologically ordered Merkle tree in CT, CTRT introduces a second tree called LexTree, where all leaf nodes are ordered lexicographically. Ryan shows that consistency proofs between the old and the new tree can be done through random checking, which is sufficient to detect cheating in practice. The advantage of the LexTree representation is that this tree allows for proofs of absence as well as proofs of currency. Those proofs of currency can then be used along CT's proof of inclusion.

Thus, the only benefit CTRT adds to CT is *Built-In-Revocation.*

3.6.5 Sovereign Keys

Sovereign Keys (SK) is a system developed by Eckersley [39]. Both Certificate Transparency and Sovereign Keys share the idea of a central repository of certificates, called "log server" in the former and "timeline server" in the latter. Unlike Certificate Transparency, the timeline servers are just flat files. The append-only property can still be guaranteed through mutual checking in a network of timeline servers. However there exists no short proof that a certificate is in the timeline server as it would in Certificate Transparency. This means that the timeline server would need to be queried every time a certificate needs verification.

As Sovereign Keys and Certificate Transparency share a similar design, many benefits are the same, hence we focus on the differences. Since the client needs to contact the timeline server every time, SK does not have *Negligible-Communication-Overhead* and relies on *Out-Of-Band-Connections.* It is therefore also not *User-Privacy-Preserving.* Yet, this also means that SK has *Built-In-Revocation,* because certificates can be revoked through a special entry in the timeline server. It is not *Resilient-To-DOS-Attacks,* because blocking the timeline servers hinders the verification process.

3.6.6 TACK

Trust Assertions for Certificate Keys (TACK) [40] is a SSL extension suggested by Moxie Marlinspike. As in standard SSL, the server generates a SSL key pair for securing the transfer channel. Additionally, the server generates a TACK key pair and signs the public SSL key with it. The idea behind the TACK key pair is that the SSL key pairs may change more often, but the TACK key should stay the same for a given domain. Therefore it is possible to conduct pinning on the domain / TACK key pair, called a pin.

In detail, when the client connects to the server, it requests the tack pin through a SSL extension in `client_hello`. In the `server_hello` message, the server then sends the TACK public key along with the SSL public key signed by the TACK secret key (TSK). The status of the connection can have three states: "unpinned", "pinned" and "rejected". Assuming the client connects to the server for the first time, it has never seen the pin before, and the connection is "unpinned". If it has seen the pin before, the pin is activated for a time span equal to the time interval between the first and the last pin for that host. If the client has an active pin for the domain the connection is "pinned". In all other cases the connection is "rejected". Marlinspike extends the protocol with an additional `min_generation` attribute for all pins in order to do revocation (i.e. incrementing the `min_generation` invalidates all pins with a smaller generation number).

Sharing the pins between clients and even relying on external sources for pins is entirely possible. But, as the authors state, "this will require additional protocols outside the scope of this document" [40, Section 8.2]. Because great care has to be taken in the engineering of the third party trust infrastructure in order to ensure that no client can be tricked into accepting or submitting incorrect pins, we do not consider pin sharing in the analysis of TACK.

Since TACK keys are generated by the server and clients base their trust entirely on the state of the pins, no CAs are required, and hence the system has *No-User-Cost* as well as *No-Server-Cost*. It is neither *Server-Compatible* nor *Browser-Compatible*, because the TACK public key must be transferred. However, it is *Incrementally-Deployable*, because once a client has pinned a server, the connection is safe as long as the pin is active. Both *Computational-Overhead* and *Communication-Overhead* is *negligible*, as only a small TACK public key needs to be exchanged. TACK works with *Trusted-Root-CAs*, *Custom-Root-CAs*, and *Selfsigned-Certificates*. Further, TACK has *No-Additional-Infrastructure*, because the whole communication happens exclusively between the server and the client. Thus, no *Out-Of-Band-Connection* is needed.

As for revocation, TACK differentiates between loosing "just" the credentials, which is handled gracefully without user intervention; and loosing the TACK secret key, which requires the user to accept the new TACK key manually. Therefore, TACK has *Quasi-*

Built-In-Revocation. OCSP or CRLs are not supported due to the missing infrastructure. It is *Resilient-To-DOS-Attacks* as well as *User-Privacy-Preserving*, because no other connection is required during verification.

Secure-Key-Migration is no problem since TACK supports publishing the new tack along with the old one. However, *Secure-Key-Migration-After-Credential-Theft* means, in the case that the TACK secret key got compromised, that the user needs to trust the new TACK key pair. The same argumentation holds for *Secure-Domain-Migration.*

TACK is the only system in our set which does not have both *First-Contact-Protection* and *Connection-Protection* since it follows the trust-on-first-use model. However, *Connection-Protection* is ensured against all levels of adversaries.

3.6.7 DANE

DNS-based Authentication of Named Entities (DANE) by Schlyter and Hoffman [41] is a system that strongly relies on the Domain Name System Security Extensions (DNSSEC). In DANE, the SSL certificate for a server is specified in the corresponding DNS record. As part of the DNS response, the client gets the certificate for the requested domain. Then the client can simply compare that certificate with the one from the server.

Of course, the security of this solution depends heavily on the security of the DNS connection. If the DNS request could be spoofed, the attacker could choose any certificate. DNSSEC has been described in detail in RFC 2535. The main issue with DNSSEC is the incomplete and slow moving deployment. Although over 100 TLDs already deployed DNSSEC, very few ISPs have done the same. Yet, only after DNSSEC is fully deployed does it offer proper protection.

As DANE is going to be integrated in the DNS services, it has *No-User-Cost* as well as *Quasi-No-Server-Cost*, since DNS providers charge for their service. DANE is *Server-Compatible* but not *Browser-Compatible*, as fetching the certificate is done by the client. It is not *Incrementally-Deployable*, since DANE requires that every domain is DNSSEC-ready, which itself is not incrementally deployable (and pending for a long time). DNS features both *Negligible-Communication-Overhead*, *Negligible-Computational-Overhead*, but only *Quasi-No-Additional-Infrastructure* because of the DNSSEC deployment requirement. The features *Trusted-Root-CAs*, *Custom-Root-CAs* and *Selfsigned-Certificates* are present, because it does not matter if the certificate is trusted as long as it comes from the DNSSEC secured DNS server. Finally, DANE has *No-Out-Of-Band-Connection*: Although the DNS query can be viewed as an out-of-band connection, we award the benefit since in most cases this connection it is necessary anyway.

DANE has no *Built-In-Revocation*, but is *OCSP-or-CRL-Compatible*. It is *Resilient-To-*

DOS-Attacks, because the only exposed infrastructure are the DNS servers. While the DNS server can of course be DOSed, it would also mean that the name resolution failed and the connection from the client to the server would not be possible either. DANE is *User-Privacy-Preserving*, as the information about which server the client connects to gets transferred just to the DNS server which knows about the request anyway. DANE supports *Secure-Key-Migration* as well as *Secure-Domain-Migration*, as the certificates can be rolled-over through the DNS server. DANE is secure (*First-Contact-Protection* and *Connection-Protection*) against level 1 and 2 adversaries, but falls when the adversary manages to compromise the DNS server.

3.6.8 AKI

Accountable Key Infrastructure (AKI) is a system proposed by Hyun-Jin Kim et al. [44]. They introduce a new party, the Integrity Log Server (ILS). Server certificates need to be registered with the ILS, and can contain signatures from multiple CAs. The ILS checks if the registration can proceed by through multiple conditions. For example, if the domain has been registered before, the new certificate must either be signed by the old key or have been signed by a larger number of CAs, optionally from a custom configured list of CAs. Users can configure many parameters of the registration process through X.509 extensions in their server's certificate.

AKI shares many properties with CT derived from the benefit of having a central certificate log. AKI is not *Browser-Compatible*, since the clients need to check for the ILS proofs. It supports *Custom-Root-CAs* through a X.509 extension that allows the user to specify a list of trusted CAs for the certificate. The biggest benefit compared to CT is that in AKI, an attacker is required to compromise n trusted parties (CAs), where n can be configured by the user, i.e. the owner of the system.

3.6.9 ARPKI

Attack Resilient Public-Key Infrastructure (ARPKI) [45] is inspired by AKI's design and employs some of its concepts. ARPKI improves AKI in several ways (c.f. [45, Section 4.6]), including a more mature implementation and a formal verification. However, none of these changes add or take any benefits with respect to our evaluation system.

3.6.10 Summary of The Evaluation

In Table 3.1 all the benefits of the evaluated systems are listed to allow for a quick comparison. While the most of the benefits are listed only once per system, the connection protection benefits are split up into three subgroups depending on the capabilities of the adversary. In the first row (➡), the adversary is only assumed to have the ability to execute an active MITMA against the client and the server. In the second row (➤), the adversary is also capable of generating an arbitrary certificate that is signed by a trusted root-CA. In the third row (➤x➡), the adversary can additionally compromise *n* parties involved in the verification process, excluding the client and the server. Black arrows indicate that this step is required for a successful man-in-the-middle attack, white arrows indicate that this step is not necessary.

As the status quo, SSL with CA-PKI scores high in the deployability benefits. However, the security is quite low: as soon as an attacker has a valid certificate for the domain, all defensive measures can be circumvented. Perspectives and Convergence have a much stronger security guarantee by requiring the attacker to compromise *n* notaries chosen by the client. Their critical weakness is that additional infrastructure is required during the validation, infrastructure that incurs a communication overhead and may be blocked or targeted by attacks. Certificate Transparency solves the problem of out-of-band connections nicely using a Merkle tree as a log server and embedding the proof that the log server saw the certificate directly into the certificate. However, this does not solve the problem of revocation: If there is no out-of-band connection, revocation relies on irregular updates. Sovereign keys is very similar to CT however it requires an online connection during every validation thus offering revocation but incurring the same problems Perspectives and Convergence have.

Critically both CT and Sovereign Keys currently do not allow custom trusted CAs to be used. This is particularly problematic for businesses wishing to operate their own CA. Choosing a completely different approach, TACK relies on the client pinning the certificate (more precisely the TACK key). This approach features strong deployability and security benefits, but with the disadvantage that there is no protection when connecting for the first time. Last but not least, DANE promises very strong security benefits. Yet, the deployment of DANE depends on the deployment of DNSSEC, which is only progressing slowly. A continuous survey by the US National Institute of Standards and Technology (NIST) shows that only approximately 1 % of large US industries completely deployed DNSSEC[3].

3.7 Open Problems

As can be seen none of the surveyed systems offer a clean sweep of the benefits. We see multiple open problems on the road to a more secure PKI for TLS.

[3] `http://fedv6-deployment.antd.nist.gov/cgi-bin/generate-com`

Incremental Deployability of TLS validation systems An important focus of further research should be on solutions that support incrementally deployment. Certificate Transparency is a good example, since browsers can simply have a whitelist of domains and/or CAs where they expect SCTs and thus allow for gradual upgrading. It is worth exploring if similar venues exists for the other validation systems discussed here.

True man-in-the-middle protection for Certificate Transparency Many of the design decisions for Certificate Transparency were made to fulfill the tight constraints and requirements of CAs and HTTPS server administrators, e.g. the need to instantly recover from key loss or the CAs requirement to be able to issue certificates without delay. This comes at the cost of having no true man-in-the-middle attack protection for CT. Fraudulent CA behavior will be detected after the fact, but there will be a time window in which end users can be attacked without defense. AKI and ARPKI do protect the end users, but don't satisfy the CAs requirements of issuing certificates without delay. Exploring this trade-off is still open research.

Industry feedback for TLS validation systems Finally, the research community should engage with the industry in discussion about a future TLS validation system. CAs, CDNs or large websites are important stakeholders and have requirements that are hard to deduce from academic discussions alone.

3.8 Conclusion

The quest to strengthen the certificate authority infrastructure and remove the weakest link nature of the system has produced a fair number of different solutions. In this chapter we introduced an evaluation framework focusing on the deployment issues and evaluated Perspectives, Convergence, Certificate Transparency, Sovereign Keys, TACK, AKI, ARPKI and DANE. We discussed the different merits and problems of these systems. Our evaluation framework can be helpful for researchers to hone in on open problems and combine beneficial elements of different systems in their quest to improve SSL certificate validation and get that system deployed.

CHAPTER 4

On Removing Unused Certificate Authorities From Trust Stores

Whereas the previous chapter focused on improving the security of the SSL/TLS ecosystem by replacing or adding parts of the validation logic, in this chapter we look into a way to improve the security properties while waiting for one of the solutions to gain traction. While there are several proposals how the CA system could be improved or replaced, none of these solutions is receiving widespread adoption yet, and even in a best case scenario it would take years to replace the current system.

In this chapter we examine a root problem of the weakest-link property and propose a simple stop-gap measure which can improve the security of HTTPS immediately. Currently, over 400 trusted entities are contained in each of the common trust stores of various platforms and operating systems. To find out which of these trusted root certificates are actually needed for the HTTPS ecosystem, we analyzed the trust stores of Windows, Linux, MacOS, Firefox, iOS and Android, discuss the interesting differences and conduct an extensive analysis against a database of roughly 47 million certificates collected from HTTPS servers. We found that of the 426 trusted root certificates, only 66 % were used to sign HTTPS certificates. We discuss the benefits and risks involved in removing the other 34 % of trusted roots. On the whole, we argue that this removal is an important first step to improve HTTPS security.

The work in this chapter was published at the International Conference on Financial Cryptography and Data Security [46] together with Sascha Fahl and Matthew Smith, both of who provided useful pointers in discussions. The rest is my own work.

4.1 Introduction

Although the collection of trusted CA certificates, called *trust store*, can in theory be configured by the user, it is de facto the operating system and browser vendors that issue the trust in the CAs. And while there is a broad consensus for a set of *common* CAs that are trusted by all common vendors, all vendors trust additional *uncommon* CAs that are not trusted by other vendors. Particularly in light of recent spying revelations, the inclusion of these uncommon CAs should be analyzed and if possible unneeded CAs should be removed.

This is a common-sense step which, surprisingly, is not actively being pursued by any of the companies responsible for the decisions on who we trust. There is a very small community of power-users who manually remove CAs they think they do not need and some tutorials on how this can be done, however, the decision on which CAs should be removed is based on anecdotal evidence and gut instinct.

As we will show in the course of this chapter, a broad majority of HTTPS servers use only CA certificates which are in all major trust stores to sign their server certificate. This makes perfect sense: Only by using a CA trusted by all platforms can a server administrator be sure that no user receives warning messages. In contrast, an adversary may be fine with an attack working only under, e.g. Windows. Therefore, those uncommon CA certificates are still a security threat.

This is especially true since an attacker could identify the client's platform by analyzing the choice and order of supported cipher suites in the TLS handshake. If those match a vulnerable platform, a MITM attack is launched; otherwise the connection would be forwarded to the legitimate server. Such an attack could go undetected for a very long time. Additionally, a CA that is present only in a few trust stores may not be subject to as much rigorous auditing as a common CA.

In this chapter we conduct a scientific analysis of which CAs are trusted on which platforms and correlate this data with 48 million certificates from Durumeric et al. that were collected by periodically scanning port 443 using ZMAP [47]. Based on this analysis, we identify 148 CA candidates that are never used to sign HTTPS server certificates. Following an in-depth analysis of these certificates, we create a list of CAs that can be removed from users' trust stores without hampering their everyday Internet activities while significantly reducing the attack surface against them. While this reduction of attack surface does not replace the need to find an improved certificate validation strategy, it is a very simple and extremely low cost measure which can be applied with minimal effort and should thus be considered as a first step to improve the security of SSL. We evaluate our reduced set of trust against two months' worth of traffic analysis in our university's network and show that there were no cases in which our proposed improvements would have caused any problems to ours users.

4.1.1 Outline

In Section 4.2 we highlight previous and parallel efforts to making SSL and the CA-PKI more secure. Section 4.3 describes our technical setup. In Section 4.4 we show which trust stores include which and how many certificates as well as how many certificates are present in every major trust store. Based on those findings, we propose a set of 140 CA certificates that can be removed from trust stores in Section 4.5. Section 6 concludes the chapter and outlines future work.

4.2 Related Work

There have been various approaches and attempts to improve the CA-PKI system. Perspectives [36] and Convergence [37] use network perspectives and multi-path probing to validate certificates and were suggested as a way to replace CAs completely. Both approaches need an additional network connection, which significantly impacts performance during connection establishment. Other approaches like Certificate Transparency [48], Sovereign Keys [39], or AKI [44] aim to control the PKI by keeping track of which CA issued which certificate. TACK [40] combines pinning with elegant key rollover. Finally, DNS DANE [49] focuses on putting certificates directly in the DNS record. While elegant, this requires the roll-out of DNSSEC, which also suffers from adoption problems [50]. All of these approaches fundamentally change the way validation is done in TLS. However, the deployment of such a new system is a huge effort. In this chapter, we focus on improving the security of the CA-PKI on the short term, offering solutions that can be deployed today to provide additional security benefits to individual users immediately.

Akhawe et al. [51] looked at click-through rates for SSL warning messages in web browsers and found that users ignore one quarter to one third of all validation errors. Based on a large dataset of TLS handshakes, Akhawe et al. [52] aimed to reduce the number of warning messages that are due to configuration or administration errors. By relaxing the validation algorithm, i.e. allowing a certificate that was issued for a certain domain to also be used for the *www* sub domain they were able to reduce the number of warnings the end user has to deal with.

In a related effort to reduce the trust put into CAs, Kasten et al. [53] analyzed which CAs usually sign for which TLDs and suggest restricting CA signing capabilities based on their signing history. They show that this can be effective, however, their system also requires some fundamental changes to the CA system.

4.3 Technical Setup

In order to evaluate which CAs could potentially be removed, we ran extensive analyses and simulations to assert that our recommendations would not lead to false positive SSL warnings. We used two different data sets for the analysis: a collection of certificates from Internet-wide ZMAP scans (the *ZMAP database*) [47], as well as all CA certificates found in trust stores (the *trust store database*). Additionally, we used a collection of two months' worth of TLS handshakes collected in our university's network in order to assert that the reduced set of CAs is still capable of validating all certificates our users encounter.

The ZMAP database consists of approximately 48 million certificates collected in periodical scans of port 443 in 2012 and 2013. For each certificate in the ZMAP database, the chain from the leaf certificate to a self-signed root was rebuilt by validating the signature of the child certificate with the parent's public key. This step was important as, according to RFC 5280 [54], HTTPS servers only need to supply intermediate CAs, not the trusted root CA. With the reconstructed chain, our dataset is independent of the server administrators' configurations.

For the trust store database, we scraped certificates from twelve trust stores used in smart phone operating systems (Android, BlackBerry, iOS), Linux distributions (CentOS, Debian, Gentoo, openSUSE, Ubuntu), as well as Mozilla Firefox, OpenBSD, OS X and Windows 8. Google Chrome does not have a trust store of its own but rather uses the trust store of the underlying operating system. Since Apple has the same policies for iOS as for OS X, both of those trust stores contain the same CA certificates. Table 4.1 shows the size of the trust stores we analyzed. Our further analysis is based on these datasets.

4.4 Trusted Root CA Certificates

The set of CA certificates included in different trust stores varies significantly. While there is a core set of 114 certificates that are included in all major trust stores (Windows, OS X, iOS, Android, Mozilla), only 28 CA certificates are present in all eleven trust stores (counting iOS and OS X as one), c.f. Figure 4.1.

	Number of certificates	Percentage
included in **1** trust store	202 certificates	47 %
included in **2** trust stores	47 certificates	11 %
included in **3** trust stores	16 certificates	4 %
included in **4** trust stores	8 certificates	2 %
included in **5** trust stores	4 certificates	1 %
included in **6** trust stores	6 certificates	1 %
included in **7** trust stores	15 certificates	3 %
included in **8** trust stores	25 certificates	6 %
included in **9** trust stores	52 certificates	12 %
included in **10** trust stores	28 certificates	6 %
included in **all** trust stores	28 certificates	6 %

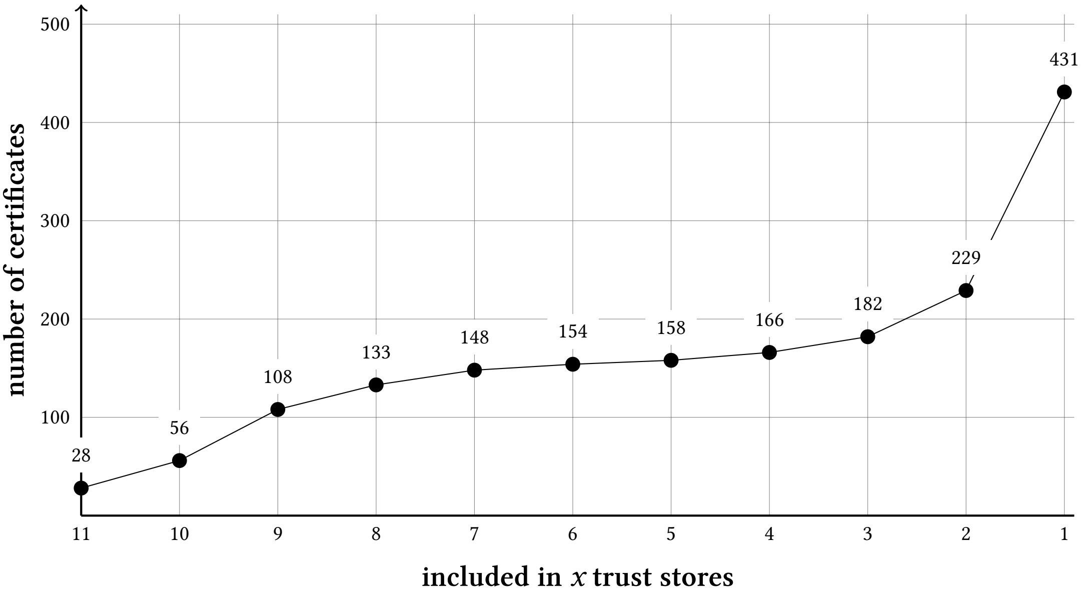


Figure 4.1: How many certificates are included in 11 (all), 10, ..., trust stores?

4.4.1 Windows Trust Store

With 377 certificates, the Windows trust store is the largest by far. Moreover, of the 202 CA certificates included in only one trust store, 168 are included only in the Windows trust store. This is partially due to the fact that the Windows trust store also contains a large number of CA certificates used for other purposes like email encryption (S/MIME) or code signing. It is possible for the Windows trust store to restrict the purpose a CA certificate can be used for, however, this is hardly done in practice. This unfortunately means that all these CAs are also trusted for HTTPS connections.

However, users may not notice how many CAs they trust, as additional CA certificates may be downloaded from the Microsoft servers as needed. Certificates can be inspected and manipulated using either the Microsoft Management Console or through the `certmgr.exe` command line tool.

4.4.2 OS X and iOS Trust Store

In OS X, administration of CA certificates is done through either the Keychain app or the `security` command line tool. Although the trust for a CA certificate can be customized to e.g. never trust the certificate for SSL, no certificate has those restrictions enabled by default. Furthermore, Apple includes their Apple Root Certificate Authority certificate in the iOS and OS X trust stores, which has never been used to sign a certificate used for HTTPS.

4.4.3 Linux/OpenBSD Trust Stores

On Linux and OpenBSD, the certificates are usually stored in a directory. By default this is `/etc/ssl/certs/`. While this makes adding and deleting certificates trivial, it is not possible to restrict the purpose of the CA certificate, for instance, to only use it for code signing.

However, the trust stores of Linux distributions are more consensus-driven: No CA certificates appear in only one trust store on these platforms. On the other side, OpenBSD is the only trust store that still includes an old CAcert Class 3 Root, while all other trust stores (that trust CAcert) include a newer CA certificate.

4.4.4 Mobile Trust Stores (Android, BlackBerry)

According to our measurements, trust stores on mobile devices tend to be both smaller in size (146 CA certificates for Android, 90 for BlackBerry), and have less unused CA

Platform	Total certs	Unused certs	To be removed	Unknown purpose	Purpose restrictable?	Restrictions used?
Windows	377	122	114	8	✓	—
Mozilla	172	23	15	8	✓	—
OS X/iOS	207	46	38	8	✓	—
Ubuntu	159	23	15	8	—	—
Debian	159	23	15	8	—	—
Gentoo	159	23	15	8	—	—
Android	146	15	7	8	—	—
openSUSE	144	14	6	8	—	—
CentOS	120	16	10	6	—	—
BlackBerry	90	14	7	7	—	—
OpenBSD	60	17	14	3	—	—
total	431	148	140	8		

Table 4.1: Used and unused CA certificates in trust stores.

certificates. Further, none of these trust stores have CA certificates that no one else trusts. This shows that it is possible to build a trust store focusing on small size and consensus while supporting all CAs needed for HTTPS.

4.4.5 Restricting the Purpose of CA Certificates

The Windows and OS X trust stores theoretically allow restricting CA certificates so that they can only be used for specific purposes like code signing, SSL, S/MIME, etc. However, we did not find any purpose-restricted CA certificates. While Windows and OS X do not use this sensible option, Linux does not offer it at all.

4.5 Removing Unneeded CAs

Roughly 34 % of all CA certificates are never used for signing HTTPS certificates. Obviously certificates could be used for other purposes and HTTPS is not the only (although most prominent) use of TLS. However: these 148 certificates can be used for signing certificates and thus for launching a MITM attack. By distrusting these CAs for SSL connections, the number of potential weakest links is reduced in a simple and straightforward manner.

Instead of removing only non-signing CAs, we further checked in how many trust stores the CAs are included. However, this only makes a difference for very few CA certificates: Of the 148 unused certificates, 140 are not included in all twelve trust stores, and 140 are not included in all major trust stores (Windows, OS X, iOS, Android, Mozilla).

Based on these results, we make two recommendations: conservative and very conservative. In the conservative recommendation, we propose that users distrust (remove/restrict) all CAs that have never signed an HTTPS certificate. This would lead to the removal of 148 CAs over all trust stores. We consider this a safe choice, since it is based on the ZMAP datasets and thus no known HTTPS certificate would create a false positive warning. Our very conservative recommendation only removes those 140 CAs which are not contained in the trust stores of Microsoft, Apple, Google, and Mozilla. Table 4.1 shows how many certificates could be removed from which trust store. While both recommendations are safe in relation to the ZMAP dataset, the very conservative recommendation is safer with respect to the possible use of a previously unused CA for signing a HTTPS certificate. However, it should be noted that especially the CAs included in all the major trust stores but have never been seen to sign an HTTPS certificate could be considered a risk factor for government coercion. The number of these CAs per trust store is listed in the *Unknown Purpose* column of Table 4.1.

4.5.1 Potential Problems and Current Solutions

Problem: False positive warnings. Removing CA certificates from the trust store could have annoying and – in the long term – potentially dangerous consequences. If users encounter a certificate that was ultimately signed by a removed CA, they will see a warning. No matter whether the users click through the warning message or stop using the site, this would encourage habituation of warning messages and further weaken the effectiveness of SSL warnings – at least for that site (c.f. [3, 55]). Therefore, when removing CA certificates, care must be taken that no legitimate certificates become invalid.

Solution. We ensured this by using a current, extensive database of HTTPS certificates that represents the current SSL landscape. Additionally, we evaluated our solution on a database of 130 million SSL handshakes and found that the proposal would not invalidate any previously valid certificates.

Problem: CA certificates are used for other purposes than SSL. As described above, our database only includes certificates for HTTPS servers. Thus CA certificates that are only used for code signing, IPSec gateways, or S/MIME would go unnoticed, be removed and could break functionality.

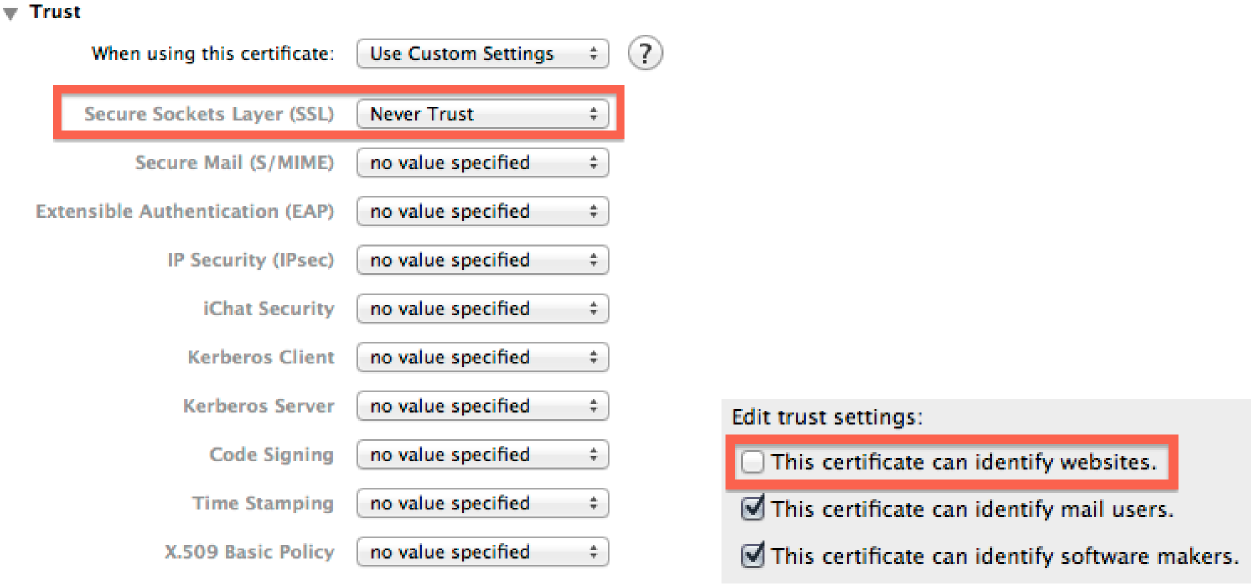
Solution. We counter this problem in two different ways. For the browser-based trust stores, there does not seem to be a reason to include CAs that do not sign HTTPS certificates, so they can simply be removed. For the Windows and OS X trust stores we recommend removing the HTTPS capabilities of those certificates (c.f. Figure 4.2). This is a conservative approach which still leaves the user open to MITM attacks for protocols such as S/MIME, however, further research is needed to determine the relevance of CAs for other protocols. Until then, breaking (non-HTTPS) SSL functionality by removing CAs too aggressively does not seem like a good idea. One caveat lurks on the Windows platform starting with Windows 7. From 7 on, Microsoft only ships a small set of CAs during the installation, but may load additional CAs on demand. This presents a unnecessary danger for the user, since it is not possible to restrict the capabilities of CAs which have not been downloaded yet. To counter this, we trigger the download of all CAs trusted by Windows and then edit the trust settings. This prevents them from being downloaded on demand with more capabilities than they need.

A critical exception to our approach is Linux which is not capable of restricting what a trusted CA can do: It is only possible to remove the CA entirely, which endangers any browser relying on the OS trust store. Interestingly, while the Google Chrome browser relies on the OS trust store on Windows and OS X, they use their own approach on Linux. The trust settings for Chrome on Linux can be configured using `certutil`, which is part of the NSS command line tools.

The potential problems for mobile devices are still work in progress. Both iOS and Android also use CA certificates for other protocols, such as RADIUS. Thus there could potentially be problems if CAs are removed solely because they have never signed a HTTPS certificate.

4.6 Conclusion

In this chapter we argued for the removal of CA certificates that do not sign any certificates used in HTTPS connections from desktop and browser trust stores. We based our analysis on an Internet-wide dataset of 48 million HTTPS certificates and compared them to trust stores from all major browser and OS vendors. We were able to identify 140 CA certificates included in twelve trust stores from all major platforms that are never used for signing certificates used in HTTPS. Based on these findings, we suggest to remove or restrict these CA certificates. Using two months' worth of TLS handshake data from our university network, we confirmed that removing these certificates from users' trust stores would not result in a single HTTPS warning message. Thus, this action provides a simple and low-cost real-world improvement that users can implement right



Trust

When using this certificate: Use Custom Settings

Secure Sockets Layer (SSL): Never Trust

Secure Mail (S/MIME): no value specified

Extensible Authentication (EAP): no value specified

IP Security (IPsec): no value specified

iChat Security: no value specified

Kerberos Client: no value specified

Kerberos Server: no value specified

Code Signing: no value specified

Time Stamping: no value specified

X.509 Basic Policy: no value specified

(a) Disabling a certificate in OS X

Edit trust settings:

□ This certificate can identify websites.

☑ This certificate can identify mail users.

☑ This certificate can identify software makers.

(b) Disabling a certificate in Mozilla Firefox

Figure 4.2: Disabling certificates for the purpose of SSL/HTTPS

now to make their HTTPS connections more secure. We are working on creating tools and scripts to automate this process for different browsers and operating systems.

Our current estimate of CAs we recommend for removal is a conservative one. It includes all CAs that have never signed a HTTPS certificate. In future work, we would like to analyze the trade-off between false positives and the size of the trust store, as well as look into mechanisms to restrict the capabilities of certificates on the Android platform.

CHAPTER 5

Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Securing the transport layer, as discussed in Chapters 3 and 4, is only one of many battles for more secure software in general. Together with Sascha Fahl et al. I have published work at CCS 2013 on making the usage of SSL more usable for Android developers [29]. Again with Sascha Fahl et al. we published at CCS 2014 work on how to make application markets' behaviour transparent, i.e. guaranteeing that updates rolled out to either all or no users, but not to only a specific subgroup.

In the same spirit of Usable Security for developers, the work in this chapter focuses on helping to audit security-critical code by limiting the amount of code that needs to be reviewed. The motivation for this is clear: the number of critical vulnerabilities as well as the amount of malware exploiting these is rising at an alarming speed. Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly.

In theory, security audits should find and remove these vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this chapter, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than current state-of-the-art systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The chapter makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results.

The work of this chapter was published as "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits" in the proceedings of the 22nd ACM CCS in 2015 [56]. Of the coauthors of this paper, Sergej Dechand, Matthew Smith and Konrad Rieck contributed through general discussions and proof-reading; Daniel Arp was responsible for the machine learning in Section 5.4; Fabian Yamaguchi contributed his experiences of analyzing C source code in order to find vulnerabilities [57]; Sascha Fahl, who provided fruitful discussions and ideas especially in the early stages of the project; and Yasemin Acar, who helped with the statistical analysis of the VCS features. The rest is my own work.

5.1 Introduction

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [58] as well as the dynamic analyzer Valgrind [59] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [60] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [61] help find possible security vulnerabilities. Finally, many languages have lint derivatives (jslint for JavaScript, pylint for Python) that can help code reviewers home in on potentially unsafe code.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must execute the analysis software on the entire project and then check if any of the warnings relate to the commit. This can be a considerable amount of work, especially since many tools require source code to be annotated and dynamic tests would have to be constructed in a way that triggers the commit.

Static and dynamic code analysis tools focus exclusively on the code without the context of who wrote the code and how it was committed. However, code repositories contain a wealth of metadata which can be highly relevant to the code quality. For instance, it can be seen whether a committer is new to the project or if they are one of the core contributors. It is possible to see the time of day or night at which code was submitted and to monitor the activity of development in certain regions of code. Moreover, most existing code-metric-based tools have very high false-positive rates, creating a (sometimes impossibly) high workload and undermining trust in the effectiveness of the tools. For instance, the state-of-the-art Flawfinder tool created 5460 false positives warnings for only 53 true positives on the dataset used in this work. It is intuitively clear that code reviewers who want to find 53 vulnerabilities in a set of 5513 flagged commits have a tough time ahead of them.

In this chapter, we present a classifier that can identify potentially vulnerable commits with a significantly lower false-positive rate while retain state-of-the-art recall rates. Therefore, unlike most existing tools for vulnerability finding, we don't focus solely on code metrics, but also leverage the rich metadata contained in code repositories. We specifically focus on reducing the falsely flagged commits, as we believe the high false alarm rate to be one of the main reasons why many tools are not seeing widespread adoption.

To evaluate the effectiveness of our approach, we conduct a large-scale evaluation of 66 GitHub projects with 170860 commits, gathering both metadata about the commits as well as mapping CVEs to commits to create a database of *vulnerability-contributing commits* (VCCs) and a benchmark for future research.

We conducted a statistical analysis of the VCCs and trained a Support Vector Machine (SVM) to detect them based on the combination of code metric analysis and GitHub metadata. For our evaluation we trained our classifier only on data up to December 31, 2010 and ran our tests against CVEs discovered in 2011–2014. We consider this a realistic test, since in a production environment we will train on all available data and then attempt to predict future vulnerabilities on a commit by commit basis.

In this dataset, our approach, called *VCCFinder*, produces only 36 false positives compared to Flawfinder's 5460 at the same level of recall. This is a reduction of over 99 % and significantly eases the workload of code reviewers.

While VCC can be used independently both on a commit by commit basis as well as to analyse entire projects, it is also well suited to be combined with other tools such as clang, Valgrind, Trinity etc.

5.1.1 Our Contributions

In summary, we make the following contributions in this work:

- We present VCCFinder, a code analysis tool that flags suspicious commits using a SVM-based detection model. Our method outperforms Flawfinder by a great margin, reducing the false positives by over 99 % at the same level of recall. Our methodology is suited to work on code snippets, enabling us to analyse code at the commit level and making a lightweight analysis of new code far easier than requiring a full build environment to be set up for each test. At Flawfinders level of recall VCCFinder flags 89 commits as potentially dangerous of which 53 are true positives and only 36 are false positives.
- We construct the first large-scale database mapping CVEs to vulnerability-contributing commits (VCCs). The database contains 66 GitHub projects, 170860 commits and 640 VCCs. We conduct an extensive evaluation of the methodology used to create this database to ascertain its quality as a benchmark for future research. As of now, this kind of benchmark database is sorely lacking, depriving researchers of test-data and a means for comparing the effectiveness of different approaches.
- We present a statistical analysis of a wide variety of different commit metrics and C/C++ keyword frequencies in relation to CVEs.
- We present an extensive quantitative and qualitative evaluation of VCCFinder and discuss take-aways, including, for instance that developers who work "9-to-5" are

more likely to produce vulnerable code than developers who commit code outside regular work hours; and that, from a security perspective, gotos are not generally harmful but in combination with error-handling code they are responsible for a significant number of VCCs.

5.2 Related Work

The discovery of vulnerabilities in program code is a fundamental problem of computer security. Consequently, it has received much attention in the past. In the following, we give a sample of the prior work most closely related to our approach.

Static analysis Several approaches to identify vulnerabilities based on lightweight scanning of source code have been presented in the past [e.g., 61–64]. Evans and Larochelle [64] introduced Splint, a lightweight static analysis tool for C based on programmer-provided annotations. However, creating the annotations must be done manually. Bandhakavi et al. [65] search for vulnerabilities in browser extensions by applying static information-flow analysis to the JavaScript code. Flawfinder [61] and RATS [62] scan source code for calls to certain functions commonly associated with vulnerabilities without requiring manual assistance, making them popular tools. This code-metric-based approach is well suited to find known bad practices. However, they do not find complex vulnerabilities and suffer from high false-positive rates, since not all bad-practice code leads to vulnerabilities.

Dynamic analysis Cho et al. [66] use a combination of symbolic and concrete execution to build an abstract model of the analyzed application and find vulnerabilities in several open-source projects. Yamaguchi et al. [57] provide an analysis platform offering fuzzy parsing of code that generates a graph representing code suitable to be mined with graph-database queries. This approach allows application-specific vulnerability patterns to be expressed; however, in contrast to our approach, it requires manual specification of these patterns by the analyst. Dahse and Holz [67] introduced a static analyzer for PHP that can detect sophisticated attacks against web applications. Holler, Herzig and Zeller [68] used fuzzing on code fragments to find vulnerabilities in the Mozilla JavaScript interpreter and the PHP interpreter. While fuzzing can produce good results it is not directly applicable to code contained in commits since the code is non-executable in that form.

Software metrics Several authors have proposed to employ software metrics to home in on regions of code more likely to contain vulnerabilities. For example, Zimmermann, Nagappan and Williams [69] perform a large-scale empirical study on Windows Vista, indicating that metrics such as code churn, code complexity [see 70, 71] and organizational measures allow vulnerabilities to be detected with high precision at low recall rates, while code dependency measures achieve low precision at high recall rates. However, Jay et al. [72] point out that many of these metrics may be highly correlated with lines of code. In particular, they show empirically that the relation between *cyclomatic complexity* and lines of code is near-linear, meaning that no reduction in the amount of code to read is achieved in this way.

Repository analysis There is a range of research work looking at software repositories in relation to software vulnerabilities. The most relevant with respect to our project can be divided into two groups: those that look at code metrics and those that look at metadata. For example, Livshits and Zimmermann [73] present Dynamine, which learns software patterns in Java programs based on previous commits and combines that with a dynamic code analysis.

Neuhaus et al. [74] use the vulnerability database of the Mozilla project to extract which software components have had vulnerabilities in the past and which imports and function calls were involved. They use this to predict which software components of the Mozilla Internet suite are most likely to contain more vulnerabilities. Unlike our approach, they do not use any metadata in their analysis and the results flag entire software components rather than single commits. The results are thus more generic in the sense that they can say only that one set of software components is more worth checking than others.

On the other side, work conducted by Meneely et al. and Shin et al. analyzes different code repository metadata in relation to CVEs [75–77]. Specifically, they check how features such as code churn, lines of code, or the number of reviewers from a project's repository and review system data correlate to reported vulnerabilities. They do this manually for the Mozilla Firefox Browser, Apache HTTP server and an excerpt of the RHEL Linux kernel. Unlike the work above and our work, they do not use this data to predict vulnerabilities; moreover, unlike our work, they do not combine the features but look at each separately.

Thus, our work goes beyond the above approaches in several ways. We combine both code metrics as well as metadata in our analysis and use a machine-learning approach to extract and combine relevant features as well as to create a classification engine to predict which commits are more likely to be vulnerable. In contrast to the work above, we do this for a large set of projects in an automated way instead of hand-picking features and analyzing single projects.

Machine-learning techniques Both machine-learning and data-mining have been proposed by several authors for finding vulnerabilities. For example, Scandariato et al. [78] train a classifier on textual features extracted from source code to determine vulnerable software components. Moreover, several unsupervised machine-learning approaches have been presented to assist in the discovery of vulnerabilities. For example, Yamaguchi et al. [79] introduce a method to expose missing checks in C source code by combining static tainting and techniques for anomaly detection. Similarly, Chang, Podgurski and Yang [80] present a data-mining approach to reveal neglected conditions and discover implicit conditional rules and their violations.

However, in order to identify vulnerabilities, these approaches concentrate only on features extracted from source code. In contrast, we show that additional meta information, such as the experience of a developer, are valuable features that improve detection performance.

5.3 Methodology

In this section, we describe how we created a database of commits that introduced known vulnerabilities in open-source projects and which features we extracted from the commits. We focus on 66 C/C++ projects using the version control system *Git* (see Section 5.3.2 for the list). These 66 projects contain 170860 commits and 718 vulnerabilities reported by CVEs. Readers unfamiliar with Git can find a description of the Git features used in our analysis in appendix 5.3.1.

5.3.1 Terminology

Popularized through the source code hosting platform GitHub, Git is a common choice for many open-source projects. Further, many more projects whose maintainers do not use Git still have Git mirrors on GitHub. Thus, basing our analysis on Git was a natural choice. For generality, our approach can easily be extended to other VCS either through one of the many Git import functions or through translating the primitives to the other VCS.

Here we introduce some Git terminology and primitives we use throughout this chapter.

- *commit:* In Git, a commit includes the state of the project at a certain point in its history (the "tree") along with a message as well as information about who authored and committed it.

- *diff:* A diff lists the differences between two Git trees. E.g. if one is interested in the changes introduced by a certain commit, one would compute the diff between its parent's tree and its own tree.
- *blame:* For each line in a given file lists the commit that is to "blame" for the change.
- *HEAD:* The HEAD of a Git branch references the most current version.

Strictly speaking a *(Git) repository* is only part of an *(open-source) project.* However, as we want to find bugs in the later through analysis of the former, we use the terms project and its corresponding main repository interchangeably.

5.3.2 List of Repositories

We used the following list of repositories: Portspoof, GnuPG, Kerberos, PHP, HHVM, apServer, Mozilla Gecko, Quagga, libav, Libreswan, Redland Raptor RDF syntax library, charybdis, Jabberd2, ClusterLabs pacemaker, bdwgc, pango, qemu, glibc, OpenVPN, torque, curl, jansson, PostgreSQL, corosync, tinc, FFmpeg, mosh, nedmalloc, trojita, inspircd, nspluginwrapper, cherokee webserver, openssl, libfep, quassel, polarssl, radvd, tntnet, Android Platform Bionic, uzbl, LibRaw, znc, nbd, Pidgin, V8, SpiderLabs ModSecurity, file, graphviz, Linux Kernel, libtiff, ZRTPCPP, taglib, Phusion passenger, suhosin, monkey, memcached, lxc, libguestfs, libarchive, Beanstalkd, Flac, libX11, Xen, libvirt, Wireshark, and Apache HTTPD.

5.3.3 Vulnerability-contributing Commits

In order to analyze the common features of commits that introduce vulnerabilities, we first needed to find out which commits actually introduced vulnerabilities. To the best of our knowledge, no large-scale database exists that maps vulnerabilities as reported by CVEs to commits. Meneely et al. and Shin et al. [75–77] manually created such mappings for the Mozilla Firefox Browser, Apache HTTP server and parts of the RHEL Linux kernel. We contacted the authors to inquire whether they would share this data, since we could have used that as a baseline for our larger analysis. Unfortunately, this was not possible at the time, although the data might be released in the future.

Since at this point we are only interested in CVEs relating to projects hosted on Github, we utilized two data sources as starting points for our mapping. As a first source, we selected all CVEs containing a link to a commit of one of the 66 projects *fixing* a vulnerability as part of the "proof". As a second source for *fixing* commits, we created a crawler that searches commit messages of the 66 projects for mentions of CVE IDs. To check the accuracy of our mapping we took a random sample of 10 % and manually checked the mapping and found no incorrectly mapped CVEs. This gave us a list of 718 CVEs. This list is potentially not complete since there might be CVEs that do not link to the fixing commit and which are also not mentioned in the commit messages. However, this does not represent a problem for our approach since 718 is a large enough sample to train our classifier.

We then developed and tested a heuristic to proceed from these fixing commits to the *vulnerability-contributing commits* (VCCs). Recall that we are operating on Git commits, which means that we have access to the whole history of a given project. One (appropriately named) Git subcommand is `git blame`, which, given a file, for each line names the commit that last changed the line. The heuristic for finding the commit that *introduced* a vulnerability given a commit that *fixed* it is as follows:

1. Ignore changes in documentation such as release notes or change logs.
2. For each deletion, *blame* the line that was deleted.

   Rationale: If the fix needed to change the line, that often means that it was part of the vulnerability. Note that Git diffs only know of added and deleted lines. If a line was changed, it shows up as a deletion and an addition in the diff.
3. For every continuous block of code inserted in the fixing commit, *blame* the lines before and after the block

   Rationale: Security fixes are often done by adding extra checks, often right before an access or after a function call. Initially, we did not *blame* for inserted lines at all, which then lead to a number of fixes which just inserted code and consequently did not have any blamed commit.
4. Finally, mark the commit as vulnerable that was blamed the most in the steps above. If two commits were blamed for the same amount of lines, blame both.

Our heuristic maps the 718 CVEs of our dataset to 640 VCCs. The reason we have fewer VCCs than CVEs is that a single commit can induce multiple CVEs. To estimate the accuracy of our heuristic, we took a 15 % random sample of all VCCs flagged by our heuristic (i.e. 96 VCCs) and manually checked them. We found only three cases (i.e. 3.1 %) where our heuristic blamed a wrong commit for the vulnerability. All three of the mis-mappings occurred in very large commits. For example, one commit of libtiff[1] that fixes CVE-2010-1411 also upgrades libtool to version 2.2.8. The method we propose for VCCFinder is capable of dealing with noisy datasets, so for the purpose of this work, an error rate of 3.1 % is acceptable. However, improving our blame heuristics further is an interesting avenue for future research.

[1] https://GitHub.com/vadz/libtiff/commit/31040a39

Apart from the 640 VCCs, we have a large set of 169502 unclassified commits. We name these commits unclassified, since, while no CVE points to them, they might still contain unknown vulnerabilities.

At this point we have a large dataset mapping CVEs to vulnerability-contributing commits. Our goal now is to extract features from these VCCs in order to detect further potential VCCs in the large number of unclassified commits.

Next, we describe which features we extracted from these commits.

5.3.4 Features

First we extracted a list of characteristics that we hypothesized could distinguish commits. One of our central hypotheses is that combining code metrics with GitHub metadata features is beneficial for finding VCCs. First, we test each feature separately using statistical analysis, e.g. for each feature we measured whether the distribution of this feature within the class of vulnerable commits was statistically different from the distribution within all unclassified commits.

Here is a list of hypotheses concerning metadata we started with:

- New committers are more likely to introduce security bugs than frequent contributors.
- It is good to "commit early and often" according to the Git Best Practices[2]. Therefore, longer commits may be more suspicious than shorter ones.
- Code that has been iterated over frequently, possibly by many different authors, is more suspicious than code that doesn't change often. Meneely and Williams [76] already analyzed these code churn features in their work. We integrate and combine these features below.

Table 5.1 shows a list of all features along with a statistical evaluation (cf. Section 5.3.6) of all numerical features except for project-scoped features. In the following, we discuss the features. For brevity reasons, we omit the discussion of self-explaining features here. All our analyses are based on commits. A commit can contain changes to one or more files. The metrics about files and functions are aggregated in the corresponding commit.

Features scoped by project are obviously the same for every commit in that project. However, in combination with other commit-based features, these can still become relevant.

[2] `http://sethrobertson.GitHub.io/GitBestPractices/#commit`

		mean	mean		
Feature	**Scope**	**VCCs**	**others**	U	**effect size**
Number of commits	Repository	282 171.39	103 980.95	32 143 126*	40 %
Number of unique contributors	Repository	524.99	236.90	30 528 184*	43 %
Contributions in project	Author	5 %	15 %	31 263 040*	42 %
Additions	Commit	306.19	71.54	20 215 148*	62 %
Deletions	Commit	73.93	37.46	42 983 290*	20 %
Past changes	Commit	627.17	385.53	40 715 632*	24 %
Future changes	Commit	792.46	396.63	36 261 346*	33 %
Past different authors	Commit	40.16	22.70	40 292 116*	25 %
Future different authors	Commit	136.58	51.44	29 534 644*	45 %
Hunk count	Commit	17.68	9.88	32 348 343*	40 %
Commit message [1]	Commit				
Commit patch [1]	Commit				
Keywords [2]	Commit				
Added functions	Function	6.51	1.03	28 724 694*	46 %
Deleted functions	Function	1.07	0.49	50 084 674*	7 %
Modified functions	Function	6.79	3.59	41 446 509*	23 %

[1] These features are text-based and thus not considered in the statistical analysis.

[2] See Table 5.2 for a statisical analyis of each keyword.

Table 5.1: Overview of the features and results of the statistical analysis of the numeric features. Mann–Whitney U test significant (*) if $p < 0.00059$.

Features Scoped by Project

Programming language The primary language the project is written in, as determined by GitHub through their open-source linguist library. In our analysis, we focused on projects written in either C or C++. The main reason for limiting our focus to one language was that we wanted to ensure comparability between the features extracted from the commit patches. When mixing different languages and syntaxes, this can't be ensured. We chose C and C++ specifically since many security-relevant projects (Linux, Kerberos, OpenSSL, etc.) are written in these languages.

Star count (number) The number of stars the project has received on GitHub. Stars are a user's way of keeping track of interesting projects, as starred projects show up on the own profile page.

Fork count (number) Forking a project on GitHub means copying the repository under your personal namespace. This is often the first step to contributing back to the project by then making changes under the personal namespace and sending a pull request to the official repository.

Number of commits (number) We counted the number of commits that are reachable from the main branches HEAD. The canonical main branch is "master", but some projects like bestpractical/rt use "stable" as the default branch. In those cases we used the branch set at GitHub by the maintainer of the project.

Watchers count (number) If a user watches a project, she will receive updates about the project's activity.

Number of contributors (number) The number of distinct authors that contributed to this project.

Project age (time and date) For the project age we took the date and time of the oldest commit reachable from the main branches HEAD.

Features Scoped by Author

Contributions (percentage) The percentage of how many commits the author has made in this project, i.e. the number of commits authored divided by the number of total commits.

Features Scoped by Commit

Number of Hunks (number) As a hunk is a continuous block of changes in a diff, this number assesses how fragmented the commit is (i.e. lots of changes all over the project versus one big change in one file or function).

Patch (text) All changes made by the commit as text represented as a bag of words.

Patch keywords (number) For each patch, we counted the number of occurrences of each C/C++ keyword. See Table 5.2 for a statistical analysis of the different distributions of each keyword.

Added, deleted and total lines (number) The number of lines added and deleted by the commit as well as the total number of changed lines.

Commit message (text) The commit message as text.

Commit time with zone (time and date) Information about when this commit was authored.

Features Scoped by File

Future changes (number) If the commit at hand is not the most current one, this is the number of times the file will be changed by later commits. We only use this feature for our historical analysis and not for the classifier, since this feature is naturally not available for new commits.

Past changes (number) The number of times this file has been changed prior to the current commit.

Past different authors (number) The number of different authors that have edited the given file.

Future different authors (number) Similar to "Future changes", the number of different authors that have changed the file in later commits.

Added, deleted and Modified functions (number) For each changed file, we record the number of added, deleted and modified functions.

Keyword	mean VCCs	mean others	U	effect size
if	39.00	7.82	37 013 390*	70 %
int	31.30	7.02	39 930 128*	68 %
struct	32.38	3.66	39 729 656*	68 %
return	18.76	3.60	41 342 834*	67 %
static	15.17	3.58	45 382 955*	64 %
void	12.52	4.31	63 935 365*	49 %
unsigned	8.66	1.51	64 440 969*	48 %
goto	5.92	0.43	64 798 818*	48 %
sizeof	4.37	0.78	66 764 357*	46 %
break	5.56	0.84	74 389 604*	40 %
char	6.71	2.68	93 400 907*	25 %

Table 5.2: Statistical analysis of C/C++ keywords sorted by effect size [81], Mann–Whitney U test significant (*) if $p < 0.000357$.

5.3.5 Excluded Features

As can be seen in Table 5.1, the vast majority of the features depend only on data gathered from the version control system and not from additional information on GitHub or any other platform. In fact, we left out some features that were only available on some projects or for few commits since the data was too sparse to reveal anything reliable. We will briefly discuss why we excluded some features which might seem counter-intuitive.

One feature that in principle would be promising but which we did not include was *issue tracker information.* GitHub provides an issue tracker and even links texts like "fixes #123" in the commit message to the corresponding issue. However, the projects which use this feature tend to be smaller projects, while the older and larger projects for which we have a rich set of CVE data predominantly use an external issue tracker. Thus, this feature is not useful for us at this time. Although a fair number of smaller projects use this, older and bigger projects often use an external issue tracker. Unfortunately these are precicely the projects for which we also have a large number of CVEs. For example, the Linux kernel use Bugzilla[3]. Although a REST API is scheduled for version 5.0, the current stable version does not have an API yet to automatically retrieve issues for a project. Other projects like OpenSSL use a Request Tracker[4], which does supply a REST API. Still this is something that would need to be configured for each project individually. OpenSSL for example requires an account in order to see the tickets.

Another piece of information that is interesting – but unfortunately too sparse at the moment – is the content of the discussion surrounding the inclusion of a change into the main repository. For this information, features could be the length of the discussion, the number of people involved, or the mean experience (in terms of contributions) of the people involved. Projects that use GitHub's functionalities extensively often do this through "pull requests". A contributor submits a commit to his own, unofficial repository and subsequently notifies the maintainer of the official repository to pull in the changes he made. GitHub provides good support for this work flow, including the ability to make comments on a pending pull request. Although this data could be useful for the classification of commits, at this point, too few projects use this work flow to be useful.

Both issue and feature request tracking as well as discussions surround a commit are handled vastly different for each project. Older projects tend to have established their work flow long before GitHub, mostly using a self-hosted issue tracker and a mailing list to discuss changes. Even if one could extract this information from all the different setups, which is certainly possible, it is questionable whether the data is comparable from project to project. Therefore, the analysis of these features is left for future work.

[3] http://www.bugzilla.org/

[4] https://www.bestpractical.com/rt/

5.3.6 Statistical Analysis of Features

For each numerical feature, we wanted to assess its fitness with respect to distinguishing VCCs from unclassified commits. We used the Mann–Whitney *U* test[5] in order to compare the distribution of a given feature within the set of commits with vulnerabilities against the set of all unclassified commits. The null hypothesis states that the feature is distributed independently from whether the commit contained a bug or not. If we can reject the null hypothesis, the feature is distributed differently in each set and thus is a promising candidate as input for the machine-learning algorithms.

We used the Bonferroni correction to correct for multiple testing for the 17 features we tested. Therefore, we test against the stricter significance level of 0.00059, which corresponds to a non-corrected $p \leq 0.01$ for each individual test. The date and time features (project age and commit with time zone) were converted to numerical features based on seconds that have elapsed since January 1, 1970 UTC (Unix epoch).

Features Scoped by Project

These features were attributed to the commit depending on the project the commit was taken from. Since all commits from a repository, whether containing vulnerabilities or not, have the same features, these features are too broad to actually distinguish commits. However, they can be valuable in combination with other features later on. For brevity, we do not discuss the features on their own here, though the table shows the significance testing.

The project statistics from GitHub (*star count*, *fork count*, *watchers count*, and *subscribers count*) do not provide any indication to whether commits from that repository contain bugs.

The features, *number of contributors*, *number of commits* as well as *project age* actually do have significantly different distributions with respect to CVEs. These three features are dependent: the longer a project exists, the more commits it has and the more contributors it has. This probably is also the reason that there is a correlation to CVEs - the more commits a project has the more likely it is to also contain commits which lead to CVEs.

Second, we looked at commits and attributed their projects' feature to them. Since security-relevant projects like OpenSSL "generate" far more CVEs than others it is no surprise that this creates different distributions. Therefore, on its own, this result does not bear much meaning. Bigger projects probably contain more bugs than smaller projects,

[5] The Mann–Whitney *U* test is used to test whether a value is distributed differently between two populations.

but there is no information on where to find them. Still, we included those features in the machine-learning algorithms, for some combination of project and commit scoped features could prove valuable.

Patch Keyword Features

For each commit we counted the occurrences of each of the following 28 C and C++ keywords: `bool`, `char`, `const`, `extern`, `false`, `float`, `for`, `if`, `int`, `long`, `namespace`, `new`, `operator`, `private`, `protected`, `sizeof`, `static`, `static`, `struct`, `switch`, `template`, `throw`, `typedef`, `typename`, `union`, `unsigned`, `virtual`, and `volatile`.

We then used the Mann–Whitney *U* test to find out whether the given keyword is used more or less frequently in VCCs compared to unclassified commits. Table 5.2 shows a subset of those keywords with high significance and high effect. We say that an effect is significant if $p < 0.000357$, corresponding to $0.01/28$, again accounting for a Bonferroni correction for multiple testing for the 28 keywords.

The effect size measures the percentage of pairs that support the hypothesis. For example, for the keyword `if`, the vulnerable commits contain more `if`s than the unclassified commits in 70 % of the cases. As can be seen by looking at the mean values for each distribution, if there is a statistical effect, the VCCs are more likely to contain those keywords compared to unclassified commits.

Features Scoped by Commit or File

All remaining features except for the number of deleted lines are distributed differently over VCC versus unclassified commits, with $p = 3.9 \times 10^{-6}$ the number of hunks being the least significant result. We note that the fact that a feature is distributed differently does not mean that this feature can be used to distinguish between the two sets. However, these results provide some hint as to why a machine-learning approach that uses a combination of these features can be successful.

The only feature where the difference was not significant was the number of deleted lines ($p = 4.6 \times 10^{-4}$), contrary to the number of added lines ($p = 3.9 \times 10^{-37}$), for which there is a significant difference in the distribution. When we manually looked at commits with known vulnerabilities and compared them to unclassified commits, we saw that the former often added a great deal of code, whereas the number of deleted or edited lines were the same as for unclassified commits. This finding confirms the intuition that security bugs are not commonly introduced by code edits or refactoring, but that new code is a more likely entry points for vulnerabilities. To the best of our knowledge this fact has not been used to ease the workload of code reviewers.

Text-Based Features

One of the central tenets of our work is that combining code metrics with GitHub metadata can help with the detection of VCCs. While both the code and the metadata features detailed above are “hard” numerical features, there are also a number “soft” features contained in GitHub that can be helpful. These text-based features, like the commit message, cannot be evaluated using statistical tests as above, but will be integrated into the machine-learning algorithm using a generalized bag-of-words model as we will discuss in Section 5.4.1.

In summary, we have observed that many features, whether coming directly from the commit or the version control system, are distributed differently within VCCs compared to unclassified commits.

5.4 Learning-Based Detection

The different features presented in the previous sections provide information for analyzing the search for suspicious commits and the discovery of potential vulnerabilities. As the large number of these features renders the manual construction of detection rules difficult, we apply techniques from the area of machine-learning to automatically analyze the commits and rank them so code-reviewers can prioritise their work. The construction of a learning-based classifier, however, poses several challenges that need to be addressed to make our approach useful in practice:

1. *Generality:* Our features comprise information that range from numerical code metrics to structured metadata, such as words in commit messages or keywords in code. Consequently, we strive for a classifier that is capable of jointly analyzing these heterogeneous features and inferring a combined detection model.

2. *Scalability:* To analyze large code repositories with thousands of source files and commits, we require a very efficient learning method which is able to operate on the large amount of available features in reasonable time.

3. *Explainability:* To help an analyst in practice, it is beneficial if the classifier can give a human-comprehensible explanation as to why the commit was flagged, instead of requiring an analyst to blindly trust a black-box decision.

We address these challenges by combining two concepts from the domains of information retrieval and machine-learning. In particular, we first create a joint representation for the heterogeneous features using a *generalized bag-of-words model* and then apply a *linear* Support Vector Machine (SVM)—a learning method that can be extended to provide explanations for its decisions and which is also efficient enough to cope with the large number of features which need to be analysed.

5.4.1 Generalized Bag-of-Words Models

Bag-of-word models have been initially designed for analysis of text documents [82, 83]. In order to combine both code metric based numerical features with GitHub metadata features, we generalize these models by considering a generic set of tokens S for our analysis. This set can contain textual words from commit messages as well as keywords, identifiers and other tokens from the code of a commit. In particular, we obtain these tokens by splitting the commit message and its code using spaces and newlines. Furthermore, we ignore certain tokens, such as author names and email addresses, since they might bias the generality of our classifier and could compromise privacy.

Formally, we define the mapping φ from a commit to a vector space as

$$\varphi : X \longrightarrow \mathbb{R}^{|S|}, \quad \varphi : x \longmapsto \big(b(x, s)\big)_{s \in S},$$

where X is the set of all commits and $x \in X$ an individual commit to be embedded in the vector space. The auxiliary function $b(x, s)$ returns a binary flag for the presence of a token s in x and is given by

$$b(x, s) = \begin{cases} 1 & \text{if token } s \text{ is contained in } x \\ 0 & \text{otherwise.} \end{cases}$$

To also incorporate numerical features like the author contribution into this model, we additionally convert all numerical features into strings. This enables us to add all arbitrary numbers to S and thereby treat both kinds of features equally. However, when using a string representation for numerical features we have to ensure that similar values are still identified as being similar. This is obviously not the case for a naive mapping, as "1.01" and "0.99" represent totally different strings.

We tackle this problem by mapping all numerical features to a discrete grid of bins prior to the vector space embedding. This quantization ensures that similar values fall into the same bins. We choose different bin sizes depending on the type of the feature. If the numerical values are rather evenly distributed, we apply a uniform grid, whereas for features with skewed distribution we a apply a logarithmic partitioning. For the latter, we apply the logarithmic function to its values and cut off all digits after the first decimal place.

To better understand this generalized bag-of-words model, let us consider a fictitious commit x, where a patch has been written by a user who did not contribute to a project before. The committed patch is written in C and contains a call to an API function which is associated with a buffer write operation. The corresponding vector representation of the commit x looks as follows

$$\varphi(x) \mapsto \begin{array}{cl} \left(\begin{array}{c} \cdots \\ 1 \\ 0 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{array}\right) & \begin{array}{l} \texttt{...} \\ \texttt{AUTHOR\_CONTRIBUTION:0.0} \\ \texttt{AUTHOR\_CONTRIBUTION:10.0} \\ \texttt{...} \\ \texttt{buf\_write\_func();} \\ \texttt{some\_other\_func();} \\ \texttt{...} \end{array} \end{array}$$

The two tokens indicative of the commit are reflected by non-zero dimensions, while all unrelated tokens are associated with zero dimensions. Note that the resulting vector space is high-dimensional and may contain several thousands of dimensions. For a concrete commit x, however, the vast majority of these dimensions are zero and thus the vector $\varphi(x)$ can be stored in a sparse data structure. We make use of the open-source tool Sally [84] for this purpose, which implements different strategies for extracting and storing sparse feature vectors.

5.4.2 Classification and Explainability

While in principle a wide range of methods are available for learning a classifier for the detection of vulnerability contributing commits, only few methods scale with larger amount of data while also providing explanations for their decisions. One technique satisfying both properties are *linear Support Vector Machines* (SVM). This variant of classic SVMs does not apply the kernel trick for learning, but instead directly operates in the input space. As a result, the run-time complexity of a linear SVM scales linearly in the number of vectors and features.

We implement our classifier for commits using the open-source tool LibLinear [85] that provides different optimization algorithms for linear SVMs. Each of these algorithms seeks a hyperplane w that separates two given classes with maximum margin, in our case corresponding to unclassified commits and vulnerability-contributing commits. As the learning is performed in the input space, we can use this hyperplane vector w for explaining the decisions of our classifier.

By calculating the inner product between $\varphi(x)$ and the vector w, we obtain a score which describes the distance from $\varphi(x)$ to the hyperplane; that is, how likely the commit introduces a vulnerability,

$$f(x) = \langle \varphi(x), w \rangle = \sum_{s \in S} w_s \, b(x, s).$$

As this inner product is computed using a summation over each feature, we can simply test which features provide the biggest contribution to this distance and thus are causal for the decision.

Finally, to calibrate free parameters of the linear SVM, namely the regularization parameter C and the class weight W, we perform a standard cross-validation on the training data. We then pick the best values corresponding to a regularization cost $C = 1$ and a weight $W = 100$ for the class of suspicious commits.

5.5 Evaluation

We evaluate the effectiveness of our approach in several different ways. First, we use a temporal split between the training and test data to evaluate the predictiveness of the SVM. We picked 2011 as the split data to have the relation of two-thirds to one-thirds training vs test data.[6] Since we have the ground truth for the years 2011 to 2014 this method allows us to realistically and reliably test the effectiveness of VCCFinder.

Note that if we would choose a random split, as is common for other machine-learning problems, a fixing commit could end up in the training set, while the corresponding VCC is contained in the testing set. The SVM would then be biased towards finding VCCs for fixes it already knows about, which is useless for actually finding new VCCs. Table 5.3 shows the distribution of commits, CVEs, and VCCs.

Second, we discuss the features learnt by the SVM as well as the true positives, i.e. vulnerabilities our classifier found in the test set. Third, we discuss the commits that are flagged by our classifier but lie outside the ground truth we have based on the CVEs. These could either be false positives or point to previously undetected vulnerabilities. Finally, we compare our approach to Flawfinder, a state-of-the-art open-source static code analyzer.

[6] This is a standard approach to evaluate classifiers. The first dataset contains all commit data up until the 31st of December 2010. We use this dataset for the design and training of our classifier. The dataset can be considered the 'historical' dataset. The second 'testing' dataset contains all commit data from 2011 to 2014 which is then used to evaluate our approach. This simulates VCCFinder being used in the beginning of 2011 having being trained on all existing data at the time and then trying to predict the unkown VCCs of the future (2011 to 2014).

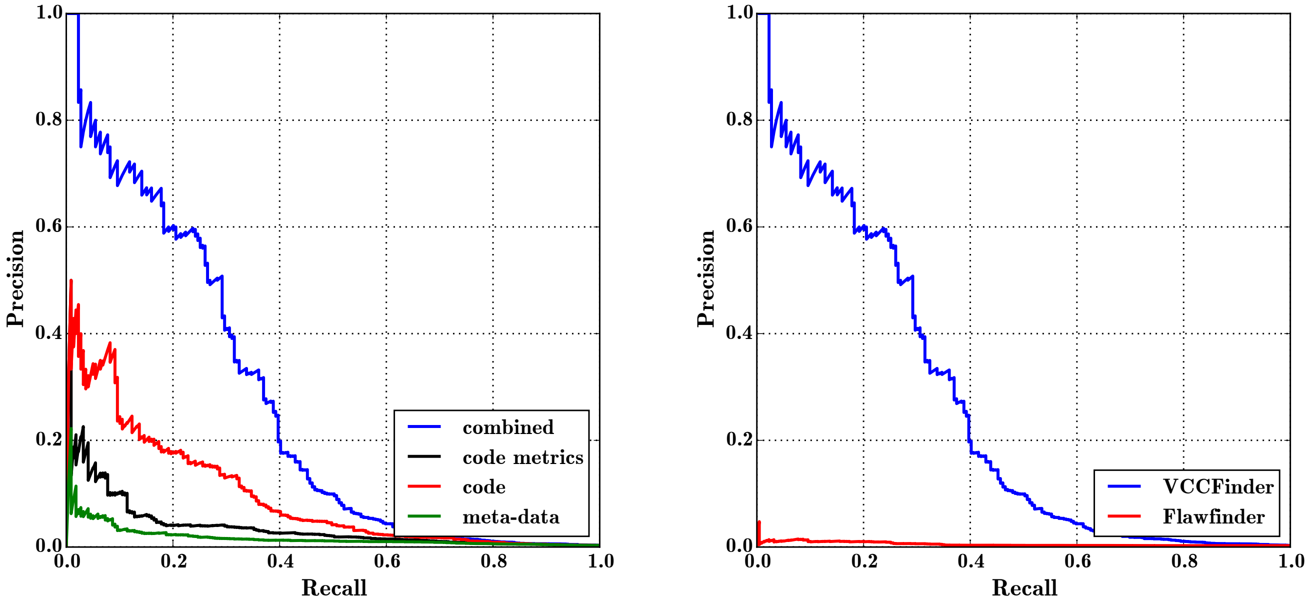
	Dataset		
	Historical	Test	Total
CVEs	469	249	718
VCCs	421	219	640
Unclassified commits	90 282	79 220	169 502

Table 5.3: Distribution of commits, CVEs, and VCCs.

Comparing feature sets We start with an evaluation of the impact of different feature sets and their combination on the detection performance of our classifier. Figure 5.1(a) shows the precision-recall curves for these experiments. To this end, we train a classifier on code metric features and meta-information. As can be seen, the classifier that combines all the features (shown in blue) out-performs the classifiers which only operate on a sub-set of the features, showing that combining the different features is beneficial. Figure 5.1(b) shows the precision recall curve of our VCCFinder compared to Flawfinder, which only operates on code metrics. The comparison with Flawfinder will be discussed in greater depth in section 5.5.3. Although the code metrics alone already provide a good detection performance, the overall precision can still be improved by using the metadata obtained from GitHub, raising the precision from 0.46 to 0.57 at a recall of 0.25. The result clearly shows that additional metadata contains useful information that can be used to improve the detection performance of our classifier.

To get further insights into these result, we inspect features that provide the biggest contribution to the detection of VCCs using the meta-information classifier. We find that often authors contributing to a project the first time and adding large amount of lines are often related to vulnerabilities. Moreover, commits where the developer differs from the actual submitter, which is indicated by a "signed-off" token, are flagged as being suspicious. Intuitively this combination of features describes the phenomena, that code written by inexperienced developers often contains vulnerabilities, but is submitted to a project without being carefully checked in advance.

Although this experiment was conducted on GitHub data in retrospective, since we split the data training on everything up to 2010 and tested it on data from 2011 to 2014, it clearly demonstrates that our classifier is capable of automatically flagging commits that are likely to contain vulnerabilities and thus would help a developer in narrowing in on flaws in their code.



(a) Detection performance of VCCFinder using different feature sets.

(b) Detection performance of our approach and FlawFinder as precision-recall curve.

Figure 5.1: Detection performance of VCCFinder.

5.5.1 Case Study

The previous section shows the precision of our approach for the different levels of recall. In practice, developers can simply decide how many commits they can afford (time- and cost-wise) to review and VCCFinder will improve their chances of finding vulnerabilities. For the sake of comparison with Flawfinder, we now set VCCFinder's recall to the same as that of Flawfinder (i.e. 0.24 cf. Table 5.4) and discuss some examples of the VCCs which would have been flagged by VCCFinder if it had been run from 2011 to 2014[7]. In these four years, VCCFinder would only have flagged 89 out of 79688 commits for manual review compared to 5513 commits flagged by Flawfinder. We believe this is a very manageable amount of code reviews to ask reviewers to do for a high return. Additionally, projects can increase the number of commits to review at any time. In the following, we present an excerpt of the vulnerabilities that VCCFinder found, when set at the very conservative level of Flawfinder's recall. We also discuss which features our classifier used to spot the VCCs. The full list of CVEs flagged by VCCFinder in this setting is shown in Section 5.5.4.

[7] As previously mentioned we use the years 2011–2014 as the test dataset, since we have ground truth data on which to base the discussion.

CVE-2012-2119 Commit `97bc3633be` *includes a buffer overflow in the macvtap device driver in the Linux kernel before 3.4.5, when running in certain configurations, allows privileged KVM guest users to cause a denial of service (crash) via a long descriptor with a long vector length*[8]. Considering metadata, our SVM detects this commit because of the edited file's high code churn, and because the author made few contributions to the Kernel in combination with the fact the the developer used `sockets`.

CVE-2013-0862 FFmpeg commit `69254f4628` *introduces multiple integer overflows in the* process_frame_obj *function in libavcodec / sanm.c in FFmpeg before 1.1.2 that allow remote attackers to have an unspecified impact via crafted image dimensions in LucasArts Smush video data, which triggers an out-of-bounds array access*[9]. The SVM detected that the author contributed little to the project before as well as that the commit inserted a large chunk of code at once.

CVE-2014-0148 In commit `e8d4e5ffdb` of Qemu *the block driver for Hyper-V VHDX Images is vulnerable to infinite loops and other potential issues when calculating BAT entries. This is due to missing bounds checks for* block_size *and* logical_sector_size *variables*[10]. The SVM found that the patch of the VCC included many keywords indicating errorprone byte manipulation, such as "opaque", "*bs", or "bytes".

CVE-2014-1438 In commit `1361b83a13`, *the `restore_fpu_checking` function in the file `arch/x86/include/asm/fpu-internal.h` in the Linux kernel before 3.12.8 on the AMD K7 and K8 platforms does not clear pending exceptions before proceeding to an EMMS instruction, which allows local users to cause a denial of service (task kill) or possibly gain privileges via a crafted application.*[11] The SVM detected a high amount of exceptions, a high number of changed code, inline ASM code, and variables containing user input such as `__input` and `user`.

5.5.2 Flagged Unclassified Commits

While we discussed the known true positive hits of our classifier for the years 2011 to 2014 above, we also have 36 commits that were flagged as potentially dangerous, for which we have no known CVE. These are commits that need be checked by code reviewers. We have shared our results with several code reviewing teams and will follow responsible

[8] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2119

[9] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0862

[10] https://bugzilla.redhat.com/show_bug.cgi?id=1078212

[11] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1438

disclosure in all cases, so we cannot discuss the flagged commits at this time. However, we can already talk about one vulnerability found by VCCFinder in commit d08d7142fd of the FFmpeg project, since this vulnerability was fixed in commit cca1a42653 before it ever was released. Thus, discussing the findings poses no harm to the FFmpeg project.

Commit d08d7142fd of FFmpeg introduces a new codec for Sierra Online audio files and Apple QuickDraw and was flagged in the 101 commits, but is not associated with a CVE. However, we discovered that in the newly created file libavcodec/qdrw.c, starting at line 72, the author does not check the size of an integer read from an adversary-supplied buffer.

```
for (i = 0; i <= colors; i++) {
  int idx;
  idx = BE_16(buf); /* color index */
  buf += 2;

  a->palette[idx * 3 + 0] = *buf++;
  buf++;
  a->palette[idx * 3 + 1] = *buf++;
  buf++;
  a->palette[idx * 3 + 2] = *buf++;
  buf++;
}
```

The macro BE_16() reads two bytes from the argument and returns an unsigned 16 bit integer. This means that an adversary controlling buf (e.g. through a malicious video) could address $3 \cdot 65535$ bytes of memory which will be filled by data from buf itself.

The SVM classified the commit because of raw byte manipulation, indicated by uses of "buf" as well as an inexperienced committer pushing a large chunk of code at once.

5.5.3 Comparison to Flawfinder

We compare our findings against *Flawfinder* [61] version 1.31, a state-of-the-art static source code scanner. Flawfinder is a mature open-source tool that has been under active development since 2001 and fulfills the requirements of being able to process C and C++ code on the level of commits. We have chosen Flawfinder for multiple reasons: first, it is open-source and under active development since 2001. Second, the requirements are similar to our tool, namely that it assumes the source to be written in C or C++ and it can operate on code-snippets as is needed to analyse commits, i.e. it does not require a working build environment. In contrast to a tool like Splint neither Flawfinder nor our tool depends on extra annotations. When given a source file, Flawfinder returns lines with suspected vulnerabilities. It offers a short explanation of the finding as well as a link to the Common Weakness Enumeration (CVE) database.[12] For the comparison, we run Flawfinder on each added or modified file of a commit. We then record the lines which Flawfinder flags that were inserted by the commit. Consequently, we say that Flawfinder marked a commit if it found a flaw in one of the lines the commit inserted.

We then evaluated both our tool and Flawfinder against the test dataset. Table 5.4 shows the contingency table, precision and recall for both tools. We argue that precision is the most important metric in this table and the one which should be used to compare Flawfinder and VCCFinder, as this value determines how many code locations a security researcher needs to look at in order to find a vulnerability. While a higher recall would *theoretically* mean that more vulnerabilities can be found, in practice they would be buried in a large amount of false positives. So for now, we accept that we will not find all vulnerabilities but create an environment in which it is realistic for a reviewer to check all flagged commits and achieve a decent success rate. Each row compares VCCFinder to Flawfinder with a different configuration. In the first row, we set VCCFinder's recall to that of Flawfinder's. As can be seen, VCCFinder's precision is significantly higher. Our approach improves the false positive rate by over 99 %! This is the most realistic configuration, since this configuration can be used in a real world setting. For the next comparison, we set VCCFinder's false positives to the same number as Flawfinder's. While of course the number of false positives is then prohibitively high, VCCFinder does find almost three times as many VCCs as Flawfinder. In the final comparison, we set VCCFinder's precision to Flawfinder's very poor value. While the number of false positives is prohibitively high, VCCFinder finds almost 90% of all VCCs compared to Flawfinder's 24%.

In Table 5.5 we also compare VCCFinder and Flawfinder based on their top results. In the first row, we select the top 100 flagged commits, then 500 and finally 1000. Among the top 100 commits, VCCFinder identifies 56 VCCs correctly, significantly reducing the amount of commits a security researcher would need to review before finding a commit containing a vulnerability. Compared to FlawFinder, its precision is more than 50 times higher and it already identifies more than 25% of all VCCs in the data set at this point.

VCCFinder significantly outperforms Flawfinder in all possible comparisons. Importantly, we were able to reduce the number of false positives to the point where it becomes realistic for reviewers to carefully check all flagged commits. This represents a significant improvement over the current state-of-the-art.

We would have liked to compare our approach to more alternatives; however, since

[12] `http://cwe.mitre.org/`

	True positive	False positive	False negative	True negative	Precision	Recall
Flawfinder	53	5460	166	73 760	0.01	0.24
VCCFinder						
— with same recall/true positives	53	36	166	79 184	0.60	0.24
— with same number of false positives	144	5460	75	73 760	0.03	0.66
— with same precision	185	24 288	34	54 932	0.01	0.84

Table 5.4: Comparison of the tools

	TP	FP	FN	TN	Precision	Recall
Flawfinder						
Top 100	1	99	218	79 121	0.01	0.00
Top 500	6	494	213	78 726	0.01	0.03
Top 1000	13	987	206	78 233	0.01	0.06
VCCFinder						
Top 100	56	44	163	79 176	0.56	0.26
Top 500	88	412	131	78 808	0.18	0.40
Top 1000	105	895	114	78 325	0.11	0.48

Table 5.5: Confusion matrix of the tools by top X commits.
T: True, F: False, P: Positive, N: Negative.

most research papers have not published the datasets they worked on and since their tools are not applicable to commits at the scale at which we tested VCCFinder, this was not possible. We are releasing our VCC database and results to the community, so that future researchers have a benchmark against which different approaches can be compared.

5.5.4 Full List of CVEs flagged

VCCFinder flagged the following 53 CVEs:

CVE-2014-7284, CVE-2014-7145, CVE-2014-5207, CVE-2014-5045, CVE-2014-4653, CVE-2014-3610, CVE-2014-3145, CVE-2014-2889, CVE-2014-1739, CVE-2014-0049, CVE-2013-7281, CVE-2013-6432, CVE-2013-6376, CVE-2013-4591, CVE-2013-4563, CVE-2013-4513, CVE-2013-4220, CVE-2013-4205, CVE-2013-4163, CVE-2013-4129, CVE-2013-4127, CVE-2013-4125, CVE-2013-3675, CVE-2013-3230, CVE-2013-3226, CVE-2013-2930, CVE-2013-2636, CVE-2013-2634, CVE-2013-2548, CVE-2013-1979, CVE-2013-1959, CVE-2013-1957, CVE-2013-1956, CVE-2013-1918, CVE-2013-1828, CVE-2013-1763, CVE-2013-0862, CVE-2013-0313, CVE-2012-6544, CVE-2012-6543, CVE-2012-6536, CVE-2012-4508, CVE-2012-4461, CVE-2012-3375, CVE-2012-2119, CVE-2011-4594, CVE-2011-4127, CVE-2011-4098, CVE-2011-2689, CVE-2011-2497, CVE-2011-2492, CVE-2011-2479, and CVE-2011-0999.

5.6 Take-Aways

As the results above show, VCCFinder's performance means that it can realistically be used in production environments without overburdening developers with a huge number of reviews. Since it can work on code snippets it can be used automatically when new commits come in without requiring a complex test environment.

Apart from this, we would like to present some qualitative take-aways we found while developing and evaluating VCCFinder, which can be useful even without using the tool. While some of these take-aways confirm well-known beliefs, we found it interesting to see that our machine-learning approach also came to these conclusions and backed them up with quantitative data, but also generated new insights.

Error handling is hard When looking at the features the SVM learnt by classifying VCCs, we saw that the adage "gotos considered harmful" [86] still holds true today, as amongst others the keyword `goto` and the according jump labels such as `out:` and `error:` increase the likelihood of vulnerable code. We can confirm this by looking at Table 5.2. However, we found that the SVM also flags returning error values such as `-EINVAL` as potentially dangerous. Combined with gotos, these are common C mechanisms for error and exception handling. So unlike Dijkstra's argument that gotos are harmful because they lead to unreadable code, in our context gotos are considered harmful because they frequently occur in an error-handling context. So instead of merely detecting gotos, our SVM gives *exception and error-handling* code a higher potential vulnerability ranking. Our explanation of this effect is as follows: because it is easy to miss some cases, exception handling is easy to get wrong (e.g. Apple's goto fail bug in their TLS implementation[13]). In conclusion, where there is error handling, there could be bugs. And more importantly, those bugs have far more potential to become security vulnerabilities than in other places of the source code. So it's a good idea to review each change in that area

[13] `https://www.imperialviolet.org/2014/02/22/applebug.html`

very carefully.

Variable Usage and Memory Management When examining highly ranked features of the SVM, we noticed that some memory management constructs lead to a higher vulnerability ranking. For instance, `sizeof(struct`, a high usage of `sizeof` in general, `len` and `length` as variable names occurred more often in vulnerable commits. In addition, we observed that variable names consisting of specific strings often occurred in VCCs: `buf`, `net`, `socket` and `sk`. While the presented keywords and variables alone do not lead to vulnerabilities, they may indicate more critical areas of the code.

Beware 9-to-5 coders We compared commits authored between business hours (from 9 to 5) with those authored outside of business hours (cf. Table 5.6). Overall, code written from 9 to 5 is 22% more likely to contain vulnerabilities (Pearson's $\chi^2 = 7.18$, $p = 0.007$). While the trend was not significant when limited to only new contributors ($\chi^2 = 0.66$, $p = 0.4$), it was even stronger for frequent contributors, with an 37% increase in the likelihood of containing vulnerabilities ($\chi^2 = 6$, $p = 0.014$). This suggests that frequent contributors produce fewer vulnerabilities outside regular working hours. We find this is a fascinating insight and plan to investigate it further in future work.

	VCCs	Unclassified commits	Percentage
New contributors			
— not 9-to-5	257	54 072	0.47 %
— 9-to-5	213	41 549	0.51 %
Factor			**1.08** ($p = 0.4$)
Frequent contributors			
— not 9-to-5	126	151 381	0.083 %
— 9-to-5	118	103 693	0.110 %
Factor			**1.37** ($p = 0.014$)
Everyone			
— not 9-to-5	383	205 453	0.19 %
— 9-to-5	331	145 242	0.23 %
Factor			**1.22** ($p = 0.007$)

Table 5.6: Comparison of commits authored between and outside of 9 to 5 o'clock.

Help new contributors We found that new contributors, i.e. contributors with less than 1 % of all commits in a given project, are about five times as likely to commit a vulnerability as their counterparts who frequently contribute. While new contributors authored 470 of 95,621 VCCs, or 0.49 %, frequent contributors authored only 244 of 255,074 VCCs, or 0.10 % (Pearson's χ^2: $p < 0.0001$). While this is of course also no big surprise, we hope quantifying this risk will help convince projects to introduce more stringent review policies.

Final Thoughts As both the evaluation and the take-aways show, commits have a myriad of possible reasons for being flagged as VCCs. These reasons can be code-based or metadata-based or, importantly, a combination of the two. The examples above give us an intuitive understanding of why this is. While our main recommendation is to use VCCFinder to classify potentially vulnerable commits to prioritize reviews, there also general recommendation which can be extracted from the classifier results.

5.7 Limitations

Our approach has several limitations. We selected 66 open-source projects written in C or C++ that created at least one CVE but otherwise varied in numbers of contributors, commits, or governance. We believe that applying our results to other projects using C or C++ should not threaten the validity. However, we can make no predictions on how VCCFinder performs on projects which to date have not received any CVEs. For generalization to other programming languages, the feature extraction and machine-learning will need to be re-done per language, so that the SVM does not mis-train based on differences in syntax.

We used a heuristic to map CVEs to VCCs. Our manual analysis of 15 % of these mappings showed that we have an error rate of 3.1 %. This needs to be taken into account by any project building on this dataset.

While we were able to map CVEs to VCCs, it is of course unknown how many unknown vulnerabilities are contained in our annotated database. Thus, our true positives must be considered a lower bound and the false positives an upper bound. So both VCFinder's and Flawfinder's results might be better than reported and the relation between them could change. However, since VCCFinder outperforms Flawfinder by a large margin, it seems unlikely that the outcome would change.

Our experiments demonstrate that VCCFinder is able to automatically spot commits that contribute to vulnerabilities with high precision; yet this alone does not ensure that an underlying vulnerability will be uncovered. Significant work and expertise is still necessary to audit commits for potential security flaws. However, our approach reduces the amount of code to inspect considerably and thus helps increase the effectiveness of code audits.

Construct validity. The code that constructs the data set could have bugs, resulting in inconsistent data and false conclusions. We mitigated this threat by writing unit tests for each component as well as checking the consistency of the resulting data using database constraints as well as additional test scripts. Further, we used the heuristic described in Section 5.3.3 in order to mark the VCCs. We manually validated the heuristic against 30 CVEs and found only one falsely marked VCC. Since this is only a small number, the heuristic should not significantly alter the results. And even a manual process would be prone to misclassifications.

5.8 Conclusion

In this chapter, we present and evaluate VCCFinder, an approach to improve code audits. Our approach combines code-metric analysis with metadata gathered from code repositories using machine-learning techniques. Our results show that our approach significantly outperforms the state-of-the-art vulnerability finder Flawfinder. We created a large test database containing 66 C/C++ project with 170860 commits on which to evaluate and compare our approach. Training our classifier on data up until 2010 and testing it against data from 2011 to 2014, VCCFinder produced 99% fewer false positives than Flawfinder, detecting 53 of the 219 known vulnerabilities and only producing 36 false positives compared to Flawfinder's 5460 false positives.

To enable future research in this area, we will release our annotated VCC database and results so that future approaches can use this database both as a training set and as a benchmark to compare themselves to existing approaches. The community is currently lacking such a baseline and we hope to spur more comparable research in this domain.

We see a very large amount of interesting future work. While the results are already significantly better than the state-of-the-art Flawfinder tool, we believe that we have only begun to scratch the surface of what can be ascertained by combining the different features. Further analyzing the results of the classifier will likely allow us to make more general recommendations on how to minimize the likelihood that vulnerabilities make it from the initial vulnerable commit into deployed software. Especially differences between new and frequent committers seems a promising avenue of future research. It will also be interesting to analyze the differences between the separate projects in more detail.

CHAPTER 6

Conclusion

As each chapter already features a detailed conclusion of each part's work, I will just highlight some overall take-aways from this work.

Take-Away 1: *New cryptographic primitives can help protect the users' privacy against the threats of Big Data.* As more and more data can easily be collected and aggregated, we need to look into new cryptographic primitives (such as homomorphic cryptography as introduced in Chapter 2) or apply new measurements such as differential privacy.

In detail, in that chapter, I facilitated Obfuscated Bloom filters to create a search algorithm that ensures the confidentiality of the queries as well as the result. The obfuscation parameter allows the scheme to be configured according to the concrete use case. The communication complexity of the scheme is linear with respect to the length of the search query. The computational complexity is logarithmic with respect to the size of the database, and linear with respect to both the length of the query as well as the length of the results. This means that even for large databases, the scheme is fast enough to be applied to real world use cases. Therefore, the presented technique is a vast improvement over PIR schemes which are not utilizing homomorphic cryptography. This work is also one of the first systems to use homomorphic cryptography practically. Beyond offering a solution for search with encrypted terms, this work also serves as an example of how systems can be designed to incorporate the new possibilities of Homomorphic Encryption.

Take-Away 2: *Big Data analytics on software and infrastructure can improve security.* When we apply data analysis techniques to assess software and infrastructure instead of spying on users, we can use the results to pinpoint weak points to fix.

Both Chapter 4 about removing unneeded certificate authorities from trust stores as well as Chapter 5 about the analysis of version control commits of open source projects highlight this.

In detail, in Chapter 4 I used data from internet-wide scans of port 443 containing approximately 48 million X.509 certificates in order to find certificate authorities that are contained in one or more trust store (i.e. trusted by one or more operating system or browser) but have not signed any actual certificate on public-facing websites. I found 140 of those certificate authorities which are included in all twelve major trust stores. I confirmed these findings with two months' worth of TLS handshake data from the University of Hanover. Based on this result, the logical recommendation is to disable the certificate authorities in question or at least limit their authorities such as to revoke their trust for validating certificates as part of HTTPS. It is notable that this list for removal is a very conservative one, since it only includes certificate authorities that have never signed a single X.509 certificate accessible over public HTTPS. However, one could further consider for removal those certificate authorities that only sign a small amount of certificates, or trust some certificate authorities only for certain domains. The last point is especially interesting for certificate authorities from German universities. Each major university holds a certificate authority and can technically sign certificates for any domain. However in practice, and by policy, they only sign certificates for their own subdomains `*.some-university.de`. Therefore, a university's certificate authority only ever needs trust for their own (sub)domains. This would further reduce the attack surface in case of a compromise without affecting valid usage.

In Chapter 5 I presented an approach to improve code audits by applying machine learning to a combination of code-metric analysis and metadata gathered from code repositories. The data-driven approach significantly outperforms the state-of-the-art vulnerability finder Flawfinder by producing 99 % fewer false positives than Flawfinder. While the results are already significantly better than the state-of-the-art Flawfinder tool, a lot more can be done by combining the different features. Further analyzing the results of the classifier will likely allow one to make more general recommendations on how to minimize the likelihood that vulnerabilities make it from the initial vulnerable commit into deployed software. Especially differences between new and frequent committers seem a promising avenue of future research. It will also be interesting to analyze the differences between the separate projects in more detail.

Data-driven analysis of software and security ecosystems In summary, I presented three case studies that explore the application of data analysis – "Big Data" – to system security. In that regard, the ability to automatically analyze data is not only a threat we need to defend our users' privacy against, but is more so also a discipline that helps improve security. By considering not just isolated examples but whole ecosystems, the insights become much more solid, and the results and recommendations become much stronger. Instead of manually analyzing a couple of mobile apps, we have the ability to consider a security-critical mistake in *all* applications of a given platform. We can identify systemic errors *all* developers of a given platform, a given programming language or a given security paradigm make – and fix it with the certainty that we truly found the core of the problem. Instead of manually analyzing the SSL installation of a couple of websites, we can consider *all* certificates – in times of Certificate Transparency even with historical data of issued certificates – and make conclusions based on the whole ecosystem. We can identify rogue certificate authorities as well as monitor the deployment of new TLS versions and features and make recommendations based on those. And instead of manually analyzing open source code bases for vulnerabilities, we can apply the same techniques and again consider *all* projects on e.g. GitHub. Then, instead of just fixing one vulnerability after the other, we can use these insights to develop better tooling, easier-to-use security APIs and safer programming languages.

Bibliography

[1] Alma Whitten and J Doug Tygar,
"Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0.", *Usenix Security*, vol. 1999, 1999 (cit. on p. 1).

[2] Anne Adams and Martina Angela Sasse, *Users are not the enemy*,
Communications of the ACM **42** (1999) 40 (cit. on pp. 1, 2).

[3] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri and Lorrie Faith Cranor,
"Crying wolf: An empirical study of SSL warning effectiveness",
Proceedings of the 18th Usenix Security Symposium, 2009 (cit. on pp. 2, 45, 64).

[4] Anne Adams, Martina Angela Sasse and Peter Lunt,
"Making Passwords Secure and Usable",
People and Computers XII: Proceedings of HCI '97,
ed. by Harold Thimbleby, Brid O'Conaill and Peter J. Thomas,
Springer London, 1997 1, ISBN: 978-1-4471-3601-9 (cit. on p. 2).

[5] Henning Perl, Yassene Mohammed, Michael Brenner and Matthew Smith,
"Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography",
E-Science (e-Science), 2012 IEEE 8th International Conference on, IEEE, 2012 (cit. on pp. 5, 6).

[6] Henning Perl, Yassene Mohammed, Michael Brenner and Matthew Smith,
Privacy/performance trade-off in private search on bio-medical data,
Future Generation Computer Systems **36** (2014) 441 (cit. on p. 5).

[7] Melissa Gymrek, Amy L McGuire, David Golan, Eran Halperin and Yaniv Erlich,
Identifying Personal Genomes by Surname Inference, Science **339** (2013) 321 (cit. on p. 6).

[8] Craig Gentry, *A fully homomorphic encryption scheme*,
PhD thesis: Stanford University, 2009 (cit. on pp. 6, 10).

[9] Jean-Sébastien Coron, David Naccache and Mehdi Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers", *EUROCRYPT'12: Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, Springer-Verlag, 2012 (cit. on p. 6).

[10] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky and William Skeith, *Public key encryption that allows PIR queries*, Advances in Cryptology-CRYPTO 2007 (2007) 50 (cit. on pp. 6, 8).

[11] Craig Gentry and Zulfikar Ramzan, "Single-Database Private Information Retrieval with Constant Communication Rate", *Automata*, Springer-Verlag, 2005 803 (cit. on pp. 6, 33).

[12] Eyal Kushilevitz and Rafail Ostrovsky, "Replication is not needed: single database, computationally-private information retrieval", *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, 1997 364 (cit. on pp. 6, 32).

[13] Dan Boneh, Eu-Jin Goh and Kobbi Nissim, *Evaluating 2-DNF Formulas on Ciphertexts*, ed. by Joe Kilian, Theory of Cryptography, vol. 3378, Lecture Notes in Computer Science, Springer-Verlag, 2005 (cit. on p. 8).

[14] Christian Cachin, Silvio Micali and Markus Stadler, "Computationally Private Information Retrieval with Polylogarithmic Communication", *Advances in Cryptology—EUROCRYPT'99*, Springer-Verlag, 1999 402 (cit. on p. 8).

[15] Jan Camenisch, Maria Dubovitskaya and Gregory Neven, "Oblivious transfer with access control", *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, ACM Request Permissions, 2009 (cit. on p. 8).

[16] Ran Canetti, Ben Riva and Guy N Rothblum, "Practical delegation of computation using multiple servers", *CCS '11: Proceedings of the 18th ACM conference on Computer and communications security*, ACM Request Permissions, 2011 (cit. on p. 8).

[17] Andrew Chi-Chih Yao, *How to generate and exchange secrets*, Foundations of Computer Science, 1986., 27th Annual Symposium on (1986) 162 (cit. on p. 9).

[18] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella et al., "Fairplay—Secure Two-Party Computation System.", *USENIX Security Symposium*, vol. 4, San Diego, CA, USA, 2004 (cit. on p. 9).

[19] Lior Malka, "VMCrypt: modular software architecture for scalable secure computation", *CCS '11: Proceedings of the 18th ACM conference on Computer and communications security*, ACM Request Permissions, 2011 (cit. on p. 9).

[20] Chris Mitchell and Chris, eds., *Trusted Computing*, Institution of Engineering and Technology, 2005 (cit. on p. 9).

[21] Craig Gentry, "Fully homomorphic encryption using ideal lattices", *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, ACM Request Permissions, 2009 (cit. on p. 10).

[22] Nigel P Smart and Frederik Vercauteren, "Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes.", *Public Key Cryptography*, vol. 6056, Springer, 2010 420 (cit. on pp. 10, 21, 22).

[23] Michael Brenner, Jan Wiebelitz, Gabriele Von Voigt and Matthew Smith, "Secret program execution in the cloud applying homomorphic encryption", *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, IEEE, 2011 114 (cit. on p. 10).

[24] Burton H Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM **13** (1970) 422 (cit. on p. 12).

[25] Ashish Goel and Pankaj Gupta, "Small subset queries and bloom filters using ternary associative memories, with applications", *SIGMETRICS '10: Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM Request Permissions, 2010 (cit. on p. 12).

[26] Zvika Brakerski, Craig Gentry and Vinod Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping", *ITCS '12: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ACM Request Permissions, 2012 (cit. on p. 22).

[27] Sergiu Costea, Dumitru Marian Barbu, Gabriel Ghinita and Razvan Rughinis, "A comparative evaluation of private information retrieval techniques in location-based services", *Intelligent Networking and Collaborative Systems (INCoS), 2012 4th International Conference on*, IEEE, 2012 618 (cit. on pp. 32, 33).

[28] Jeremy Clark and Paul C van Oorschot, "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements", *Security and Privacy (SP), 2013 IEEE Symposium on*, IEEE, 2013 511 (cit. on pp. 37–40).

[29] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter and Matthew Smith, “Rethinking SSL development in an appified world”, *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, 2013 49 (cit. on pp. 37, 67).

[30] Sascha Fahl, Yasemin Acar, Henning Perl and Matthew Smith, “Why eve and mallory (also) love webmasters: a study on the root causes of SSL misconfigurations”, *Proceedings of the 9th ACM symposium on Information, computer and communications security*, ACM, 2014 507 (cit. on p. 37).

[31] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek and Matthew Smith, “Hey, NSA: Stay away from my market! Future proofing app markets against powerful attackers”, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014 1143 (cit. on p. 37).

[32] EFF, *SSL Observatory*, URL: `https://www.eff.org/observatory` (cit. on p. 38).

[33] Joseph Menn, *Key Internet operator VeriSign hit by hackers*, URL: `http://www.reuters.com/article/2012/02/02/us-hacking-verisign-idUSTRE8110Z820120202` (cit. on p. 38).

[34] Comodo, *Comodo Report of Fraud Incident*, URL: `https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html` (cit. on p. 38).

[35] Jonathan Nightingale, *DigiNotar Removal Follow Up*, 2013, URL: `https://blog.mozilla.org/security/2011/09/02/diginotar-removal-follow-up/` (cit. on p. 38).

[36] Dan Wendlandt, David G Andersen and Adrian Perrig, “Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing”, *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008 321 (cit. on pp. 38, 46, 47, 59).

[37] Moxie Marlinspike, “SSL And The Future Of Authenticity”, *BlackHat USA 2011* (cit. on pp. 38, 47, 48, 59).

[38] Ben Laurie, Adam Langley and Emilia Kasper, *Certificate Transparency. Network Working Group Internet-Draft, v12, work in progress*, 2013, URL: `http://tools.ietf.org/html/draft-laurie-pki-sunlight-12` (cit. on pp. 38, 40, 47).

[39] Peter Eckersley, *Sovereign Key Cryptography for Internet Domains*, URL: `https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=master` (cit. on pp. 38, 47, 50, 59).

[40] Moxie Marlinspike, *TACK: Trust Assertions for Certificate Keys*, URL: `http://tack.io/draft.html` (cit. on pp. 38, 47, 51, 59).

[41] Jakob Schlyter and Paul Hoffman, *The DNS-Based Authentication of Named Entities (DANE) Protocol for Transport Layer Security (TLS): TLSA*, 2012, URL: `https://tools.ietf.org/html/rfc6698` (cit. on pp. 38, 47, 52).

[42] Joseph Bonneau, Cormac Herley, Paul C van Oorschot and Frank Stajano, *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes*, Security and Privacy (SP), 2012 IEEE Symposium on () (cit. on pp. 41, 42).

[43] Mark D Ryan, *Enhanced certificate transparency and end-to-end encrypted mail*, Proceedings of NDSS. The Internet Society (2014) (cit. on pp. 47, 50).

[44] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson and Virgil Gligor, “Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure”, *Proceedings of the 2013 Conference on World Wide Web*, 2013 (cit. on pp. 47, 53, 59).

[45] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse and Pawel Szalachowski, “ARPKI: Attack resilient public-key infrastructure”, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014 382 (cit. on pp. 47, 53).

[46] Henning Perl, Sascha Fahl and Matthew Smith, “You Won’t Be Needing These Any More: On Removing Unused Certificates From Trust Stores”, *International Conference on Financial Cryptography and Data Security*, Springer, 2014 307 (cit. on p. 57).

[47] Zakir Durumeric, Eric Wustrow and J Alex Halderman, “ZMap: Fast Internet-wide scanning and its security applications”, *Proceedings of the 22nd USENIX Security Symposium*, 2013 (cit. on pp. 58, 60).

[48] B. Laurie, A. Langley and E. Kasper, *Certificate Transparency*, RFC 6962 (Experimental), Internet Engineering Task Force, 2013, URL: `http://www.ietf.org/rfc/rfc6962.txt` (cit. on p. 59).

[49] P. Hoffman and J. Schlyter, *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*, RFC 6698 (Proposed Standard), Internet Engineering Task Force, 2012, URL: `http://www.ietf.org/rfc/rfc6698.txt` (cit. on p. 59).

[50] Wilson Lian, Eric Rescorla, Hovav Shacham and Stefan Savage, "Measuring the practical impact of DNSSEC deployment", *Proceedings of the 22nd USENIX conference on Security*, USENIX Association, 2013 573 (cit. on p. 59).

[51] Devdatta Akhawe and Adrienne Porter Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness", *Proceedings of the 22th USENIX Security Symposium*, 2013 (cit. on p. 59).

[52] Devdatta Akhawe, Bernhard Amann, Matthias Vallentin and Robin Sommer, "Here's my cert, so trust me, maybe?: understanding TLS errors on the web", *Proceedings of the 22nd international conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2013 59 (cit. on p. 59).

[53] James Karsten, Eric Wustrow and J Alex Halderman, "CAge: Taming Certificate Authorities by Inferring Restricted Scopes", *FC'13: Proceedings of the 17th international conference on Financial Cryptography and Data Security*, 2013 (cit. on p. 59).

[54] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley and W. Polk, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 5280 (Proposed Standard), Updated by RFC 6818, Internet Engineering Task Force, 2008, URL: `http://www.ietf.org/rfc/rfc5280.txt` (cit. on p. 60).

[55] Serge Egelman, Lorrie Faith Cranor and Jason Hong, "You've been warned", *Proceeding of the twenty-sixth annual CHI conference*, ACM Press, 2008 1065 (cit. on p. 64).

[56] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl and Yasemin Acar, "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits", *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, ACM, 2015 426, ISBN: 978-1-4503-3832-5, URL: `http://doi.acm.org/10.1145/2810103.2813604` (cit. on p. 68).

[57] Fabian Yamaguchi, Nico Golde, Daniel Arp and Konrad Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs", *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014 (cit. on pp. 68, 71).

[58] *Clang Static Analyzer*, `http://clang-analyzer.llvm.org/`, Accessed: 2015-05-08 (cit. on p. 68).

[59] *Valgrind*, `http://valgrind.org/`, Accessed: 2015-05-08 (cit. on p. 68).

[60] *Trinity: A Linux system call fuzzer*, `http://codemonkey.org.uk/projects/trinity/`, Accessed: 2015-05-08 (cit. on p. 69).

[61] David A. Wheeler, *Flawfinder*, `http://www.dwheeler.com/flawfinder/`, visited January, 2015 (cit. on pp. 69, 71, 90).

[62] *Rough Auditing Tool for Security (RATS)*, `https://code.google.com/p/rough-auditing-tool-for-security/`, visited January, 2015 (cit. on p. 71).

[63] *PREfast Analysis Tool*, `https://msdn.microsoft.com/en-us/library/ms933794.aspx`, visited January, 2015 (cit. on p. 71).

[64] David Evans and David Larochelle, *Improving security using extensible lightweight static analysis*, IEEE software **19** (2002) 42 (cit. on p. 71).

[65] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan and Marianne Winslett, "VEX: Vetting Browser Extensions for Security Vulnerabilities.", *USENIX Security Symposium*, vol. 10, 2010 339 (cit. on p. 71).

[66] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu and Dawn Song, "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery.", *USENIX Security Symposium*, 2011 139 (cit. on p. 71).

[67] Johannes Dahse and Thorsten Holz, "Static Detection of Second-Order Vulnerabilities in Web Applications", *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, 2014 989 (cit. on p. 71).

[68] Christian Holler, Kim Herzig and Andreas Zeller, "Fuzzing with Code Fragments", *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, USENIX, 2012 445 (cit. on p. 71).

[69] Thomas Zimmermann, Nachiappan Nagappan and Laurie Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista", *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, IEEE, 2010 421 (cit. on p. 72).

[70] Thomas J McCabe, *A complexity measure*, Software Engineering, IEEE Transactions on (1976) 308 (cit. on p. 72).

[71] Maurice H Halstead, *Elements of software science*, Elsevier computer science library : operational programming systems series, North-Holland, 1977 (cit. on p. 72).

[72] Graylin Jay, Joanne E Hale, Randy K Smith, David P Hale, Nicholas A Kraft and Charles Ward, *Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship.*, JSEA 2 (2009) 137 (cit. on p. 72).

[73] Benjamin Livshits and Thomas Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories", *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005 (cit. on p. 72).

[74] Stephan Neuhaus, Thomas Zimmermann, Christian Holler and Andreas Zeller, "Predicting vulnerable software components", *Proceedings of the 14th ACM conference on Computer and communications security*, ACM, 2007 529 (cit. on p. 72).

[75] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant and Kayla Davis, "An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities", *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, ACM, 2014 37 (cit. on pp. 72, 74).

[76] Andrew Meneely and Oluyinka Williams, *Interactive Churn Metrics: Socio-technical Variants of Code Churn*, SIGSOFT Softw. Eng. Notes 37 (2012) 1 (cit. on pp. 72, 74, 76).

[77] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary and Brian Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits", *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, IEEE, 2013 65 (cit. on pp. 72, 74).

[78] Riccardo Scandariato, James Walden, Aram Hovsepyan and Wouter Joosen, *Predicting vulnerable software components via text mining*, IEEE Transactions on Software Engineering 40 (2014) 993 (cit. on p. 73).

[79] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon and Konrad Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery", *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, 2013 499 (cit. on p. 73).

[80] Ray-Yaung Chang, Andy Podgurski and Jiong Yang, *Discovering neglected conditions in software by mining dependence graphs*, IEEE Transactions on Software Engineering **34** (2008) 579 (cit. on p. 73).

[81] Hans W Wendt, *Dealing with a common problem in Social science: A simplified rank-biserial coefficient of correlation based on the U statistic*, European Journal of Social Psychology **2** (1972) 463 (cit. on p. 79).

[82] Gerard Salton and Michael J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, 1986 (cit. on p. 84).

[83] Gerard Salton, *Mathematics and information retrieval*, Journal of Documentation **35** (1979) 1 (cit. on p. 84).

[84] Konrad Rieck, Christian Wressnegger and Alexander Bikadorov, *Sally: A Tool for Embedding Strings in Vector Spaces*, Journal of Machine Learning Research (JMLR) **13** (2012) 3247 (cit. on p. 85).

[85] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang and Chih-Jen Lin, *LIBLINEAR: A Library for Large Linear Classification*, Journal of Machine Learning Research (JMLR) **9** (2008) 1871 (cit. on p. 85).

[86] Edsger W Dijkstra, *Letters to the editor: go to statement considered harmful*, Communications of the ACM **11** (1968) 147 (cit. on p. 93).

List of Figures

List of Tables