Towards Empirical Evaluation of Software Security Risk

by

Jenny Blessing

B.S., Computer Science, University of Connecticut (2019)

Submitted to the Institute for Data, Systems, and Society &
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Technology and Policy
and
Master of Science in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2021

Author ....
Institute for Data, Systems, and Society &
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by..
Daniel J. Weitzner
3Com Founders Principal Research Scientist, CSAIL
Founding Director, Internet Policy Research Initiative
Thesis Supervisor

Accepted by ....
Noelle Eckley Selin
Associate Professor, Institute for Data Systems and Society &
Department of Earth, Atmospheric, and Planetary Sciences
Director, Technology and Policy Program

Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Towards Empirical Evaluation of Software Security Risk

by

Jenny Blessing

Submitted to the Institute for Data, Systems, and Society &
Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Technology and Policy
and
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis provides empirical metrics for different vectors for vulnerability introduction, with a particular focus on cryptographic software. Through quantitative analysis of source code and vulnerability metrics from a variety of cryptographic libraries, we arrive at a more precise notion of what types of modifications introduce a higher level of risk into a system. Empirical evidence of the causes of security risk will provide technically-grounded guidance in the ongoing policy debate over exceptional access, enabling the security community to more objectively evaluate proposed exceptional access systems.

Thesis Supervisor: Daniel J. Weitzner
Title: 3Com Founders Principal Research Scientist, CSAIL
Founding Director, Internet Policy Research Initiative

Acknowledgments

I owe an incredible debt of gratitude to my advisor, Danny Weitzner, and Mike Specter: thank you both for your mentorship and support. Your guidance and encouragement made me a better researcher and were instrumental to the development of this thesis.

This work would not have been possible without the tireless administrative staff of TPP, EECS, and IPRI, especially Barb DeLaBarre and Mel Robinson. I am deeply grateful for all of their help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Despite the importance of system security in a society ever more dependent on software systems, the security community lacks good standards for objectively evaluating and comparing the security of different systems. Software that implements cryptographic primitives has few standardized methods of risk assessment even as cryptography has developed cryptographic strength metrics based on assumptions of mathematical hardness.

The goal of this thesis is to explore evaluation metrics for formally characterizing security risk in a system as the complexity of modern software systems grows faster than our ability to secure them. Standard software security metrics that are scalable and can be widely applied will enable developers to make more informed choices when modifying existing code or adding new features, and will also give the technical community the tools to better appraise systems from a security standpoint.

In particular, our focus is on determining metrics for cryptographic software, which we define as software that implements cryptographic algorithms, protocols, or schemes. Cryptographic libraries such as OpenSSL and GnuTLS are among the largest open-source cryptographic software systems. These libraries are tasked with securing the modern Internet, making security risk measurements of cryptographic code of particular interest compared to general-purpose codebases.

Analyzing the causes and frequency of vulnerabilities in cryptographic libraries, we develop empirical metrics for security risk in cryptographic code. We evaluate how well common software characteristics, including as lines of code and cyclomatic complexity, correlate with security outcomes, such as vulnerability count and vulnerability lifetime. Moving away from anecdotal statements like "complexity is the enemy of security," this thesis instead asks: how much security risk does complexity introduce? How does maintenance of legacy cryptographic schemes and protocols affect the overall security of a system? And are there particular categories of cryptographic changes that cause increased security risk? The answers to these questions and others will contribute to the emerging field of evidence-based software security and help the security community to better evaluate proposed system modifications.

1.1 Motivation

Cryptographic libraries such as OpenSSL are responsible for securing virtually all network communication. Consequently, a vulnerability in any one of them threatens to compromise a significant percentage of web traffic. Despite the critical role these libraries play and our heavy dependence on them, they have consistently produced outsized quantities of vulnerabilities, some of which have been severe. One of the most infamous vulnerabilities, Heartbleed, was discovered in OpenSSL in March 2014 and enabled attackers to read the contents of private memory off of servers using affected versions of OpenSSL. At the time of Heartbleed's disclosure, over 66% of all websites were vulnerable [7]. More recently, in June 2020 GnuTLS suffered a severe vulnerability in its implementation of TLS sessions that allowed attackers to passively decrypt traffic [68].

Our focus on cryptographic software is especially relevant because of renewed government interest in regulating and modifying cryptographic code [19, 99]. This thesis will provide technical evidence useful in progressing past the current impasse in the ongoing debate over building law enforcement mechanisms for exceptional access to encrypted data, where the security community has few known metrics for objectively analyzing the risk present in a system, but at the same time, policymakers ask for a generalizable definition of what makes a system insecure. Empirical analysis of cryptographic software and vulnerability metrics will help the technical community to draw broader conclusions about system security risk and better determine what kinds of exceptional access modifications would make a system unacceptably insecure. It should be noted that while this thesis focuses solely on technical security risk, exceptional access introduces other types of risk, like operational risk, that also need to be considered in the debate.

1.2 Thesis Contributions

In this thesis, we conduct the first comprehensive analysis of cryptographic libraries and the vulnerabilities affecting them. We select eight of the most widely used cryptographic libraries and build a dataset of the approximately 300 entries in the National Vulnerability Database (NVD) [98] for these systems. Using the collected data we explore the relationship between cryptographic complexity and security and explore answers to the questions posed above, with particular attention to the question of feature trade-offs: when should a developer make a cryptographic change or keep a security feature around, and when is it not worth the security risk?

Our findings include the following: just 27.24% of vulnerabilities in cryptographic software are cryptographic issues, while 37.2% of errors are related to memory mismanagement or misuse. Vulnerabilities can live in the software for a long time: the average exploitable lifetime of a vulnerability in a cryptographic library is 4.06 years. In OpenSSL specifically, we find that at least 1 vulnerability is introduced for every thousand lines of code added.

We further investigate whether trends observed are unique to cryptographic source code, collecting similar metrics from systems that are non-cryptographic in nature, such as browsers or operating systems. As we describe in Chapter 5, we find that the rate of the rate of vulnerability introduction is up to three times as high in cryptographic as in non-cryptographic software.

In addition to the quantitative study of cryptographic software and its vulnerabilities, this thesis further presents a systematic review of proposed modifications to encrypted systems for law enforcement access, referred to as "exceptional access" mechanisms. We provide high-level summaries of each of the contemporary proposals for exceptional access, a brief discussion of their similarities and differences, and an in-depth discussion of how software implementation vulnerabilities could have the potential to impact these schemes.

In summary, the contributions of this thesis are:

- A formal characterization of vulnerabilities in cryptographic software
- A quantitative analysis of the relationship between complexity and security in cryptographic software
- A quantitative comparison of code complexity in cryptographic and non-cryptographic software
- A discussion of the contemporary exceptional access debate
- A qualitative analysis of technical proposals for exceptional access
- A discussion of cryptographic software risk and its applications to the exceptional access debate

1.3 Roadmap for Future Sections

We begin with a discussion of the contemporary exceptional access debate in Chapter 2, providing context and motivation for the empirical analysis done in the subsequent sections.

Chapters 3 through 5 seek to better understand the software implementation vulnerabilities introduced in cryptographic source code and their causes. First, in Chapter 3, we characterize the vulnerabilities discovered in cryptographic libraries. Chapter 4 investigates their causes through evaluating the effects of software complexity on vulnerability frequency. In Chapter 5 we compare our findings with non-cryptographic systems, assessing whether increased software complexity affects the security of cryptographic software more than non-cryptographic software.

In Chapter 6, we discuss technical proposals enabling access since these proposals serve as real-world examples of proposed cryptographic modifications introducing some unknown level of risk. Chapter 7 concludes the thesis with a discussion of takeaways for both software engineering practices and the encryption debate. Taken together, this thesis evaluates several cryptographic codebases on the basis of software and security metrics, demonstrating the security posture of common cryptographic libraries and providing a close approximation to the behavior of exceptional access source code in modern software.

Chapter 2

Overview of Exceptional Access Debate

For decades, law enforcement, computer scientists, and civil libertarians have quarreled over the spread of encryption and whether technology companies should build mechanisms to allow for government access to encrypted data. The contemporary debate over encryption and law enforcement surveillance is highly polarized, with government officials and technologists both advocating absolute stances. Both of the two extremes—unrestricted secrecy and unrestricted access—are liable to cause harm to national security and civil liberties, respectively. The technical dimensions of the debate have centered around security, with tension between computer scientists and law enforcement over how to balance national security needs with general system security requirements [48, 89].

We cannot have a productive debate or make progress towards a technically-grounded balance, though, without a more concrete understanding of the potential security consequences of building an exceptional access capability for law enforcement. In testimony offered at various Congressional hearings on the subject, witnesses warning of technical security implications relied on high-level characterizations of cryptographic systems as "among the most fragile and subtle elements of modern software" and testified that an exceptional access mechanism would introduce "unreasonable" security risk [39, 35]. While it is commonly accepted in the technical community that the addition of an exceptional access capability lowers the security of a system by some amount, the particular security cost has not been clearly characterized, and so the trade-off between user security and national security is not well understood.

2.1 Law Enforcement Concerns over End-to-End Encryption

Up until as recently as 2014, most data on phones and other personal devices was stored in an unencrypted form. As long as device manufacturers did not encrypt devices by default, the data was readily accessible by law enforcement via traditional legal mechanisms as evidence in an investigation. Communications data was also largely unencrypted: although *sensitive* data, such as login information, was typically encrypted for transit, most other web traffic was not [45].

In the wake of the Snowden revelations in 2013, this landscape changed. Concern over government surveillance from device users and the broader technical community prompted Apple and other manufacturers to accelerate the implementation of end-to-end encryption in their products and to begin encrypting personal devices by default [45, 8], preventing even the companies themselves from reading the data without user cooperation. Consequently, companies could no longer respond to lawful government requests for access to data, even when law enforcement obtained a warrant, for they lacked access to the data they were being asked to turn over.

Law enforcement agencies and government officials in the United States and abroad have raised concerns over this loss of access to data [83, 19]. The full consequences of increased adoption of encryption on law enforcement investigative capabilities are not fully understood, but law enforcement officials maintain they have lost access to data critical for national security and counter-terrorism investigations. In response to the loss of previous sources of data, colloquially referred to as these sources "going dark," law enforcement has primarily adopted what is known as "lawful hacking" as a workaround to gain access to the device in question. Lawful hacking refers to exploiting existing vulnerabilities in the device of which the vendor is unaware (and therefore has not patched) in order to gain access. The resolution of the FBI-Apple encryption dispute in 2016 is one such example of lawful hacking, where the FBI was able to unlock an iPhone with the assistance of a third party [19].

Law enforcement's preferred solution is to compel companies to build an exceptional access mechanism, allowing law enforcement to gain access but in such a way that it cannot be exploited by malicious actors [10]. Gaining access to the desired data through a third-party vendor is not always possible and can be quite financially expensive [9], making direct company access more efficient and effective.

2.2 Response from Technical Community

The recurring debates between government officials and computer scientists over the viability of building exceptional access into encrypted products are often referred to as the "crypto wars" [38]. The technical community has long maintained that it is not possible to build such an exceptional access capability without damaging the security of the system in which the capability is built [10, 13]. Supporting this stance, security researchers have demonstrated vulnerabilities in several prior attempts by governments in the U.S. and elsewhere to design communications technologies enabling access by authorized parties [84, 20]. In particular, the Clipper Chip scheme and its flaws are discussed in more detail in Section 6.

While technologists have shown particular exceptional access systems to be demonstrably insecure, they have relied heavily on anecdotal information-security maxims and provided comparatively little empirical research to substantiate their broader security arguments. Security practitioners have long maintained a correlation between complexity and security, with the phrase "complexity is the enemy of security" popularized by Bruce Schneier in a 1999 essay [88]. Moreover, cryptographic code is notoriously subtle and difficult to implement correctly, leading security researchers to be even more wary of extraneous additions or modifications to cryptographic source code. Any exceptional access capability, though, inherently adds complexity to a system, turning this common saying into a dogmatic argument in favor of refusing to consider any kind of lawful access.

Prior analysis from the technical community has concluded that cryptographers and security researchers have not demonstrated that it is possible to build a lawful access capability without lowering the security of a system by some amount. The U.S. National Academies produced a report in 1996 advocating for the continued widespread use of cryptography [29], and again in 2018 providing the most comprehensive overview of the encryption debate to date [67]. The committee's 2018 report describes the role of encryption in modern society and the consequences of increased encryption on law enforcement investigative capabilities, with a particular focus on making the debate digestible to policymakers. While the report is intended to provide a general framework for the encryption debate and does not advocate a particular viewpoint, it concludes that building an exceptional access capability would be technically challenging and that there are no contemporary technical proposals for exceptional access that are without security risks.

Abelson et al. [10] explicitly advocated against an exceptional access mandate in 2015, summarizing the potential risks, technical and otherwise, of such a mandate. The arguments provided were fairly high-level and provided little discussion of the technical details of how government access might be implemented. In the following chapters, we build on Abelson et al.'s work by providing quantifiable metrics to substantiate prior arguments from the technical community.

Chapter 3

Characterizing Vulnerabilities in Cryptographic Software

The security of the Internet rests on a small number of open-source cryptographic libraries, yet these systems notoriously suffer from large numbers of vulnerabilities. In this chapter, we perform a systematic analysis of vulnerabilities in cryptographic libraries using data scraped from the National Vulnerability Database (NVD), individual projects' GitHub repositories, internal mailing lists, project bug trackers, and various other external references. We chose to look at open-source cryptographic libraries because these systems are the most widely used software codebases where the source code is almost entirely cryptographic in nature (i.e. implements cryptographic primitives, protocols, or algorithms). Our evaluation is sufficiently in-depth for a rigorous analysis while also general enough to enable generic understanding across multiple projects.

To assess the causes of vulnerabilities in cryptographic software, we begin by characterizing various aspects of discovered CVEs and the vendor policies under which they are reported. In this chapter and subsequent ones, we define vulnerability as an entry in the Common Vulnerabilities and Exposures (CVE) list maintained by MITRE [63].

While there is a large body of existing work studying vulnerability life cycles and patches, prior work has not included cryptographic software in their datasets, despite the security-critical nature of such software. Li et al. [55] and Shahzad et al. [91] both conducted large-scale analyses of vulnerability characteristics in non-cryptographic open-source software, including various operating systems and web browsers. Rescorla et al. [85] manually analyzed a dataset of 1,675 vulnerabilities from the Linux kernel and other large software systems, defining vulnerability lifetime through window of exposure, and found inconclusive evidence that public vulnerability disclosure is worth the security implications. The unique characteristics of cryptographic software suggest that vulnerability data from non-cryptographic systems may not be applicable, necessitating a separate investigation of cryptographic software specifically.

Other studies of cryptographic libraries [66, 11, 47] have focused primarily on their usability, finding that the software and documentation of these libraries are opaque and difficult for non-specialists to understand. In particular, several studies have shown that developers overwhelmingly struggle to use these libraries correctly, with overly complex implementation and poor documentation among the root causes of this usability gap.

What prior work exists on the security of cryptography APIs has typically singled out a particular API and done an in-depth analysis of that single library. Walden [100] examined OpenSSL, historically the most widely-used cryptography API, and compared the software quality, library vulnerability count, and other metrics before and after the Heartbleed vulnerability, finding that OpenSSL had made signifiant quality improvements in Heartbleed's aftermath. Other security studies focused on less commonly used or proprietary APIs. No previous study has compared different libraries or analyzed codebase and vulnerability trends across multiple libraries.

3.1 Methodology

In this section, we discuss the systems and vulnerability databases included in the study and our reasons for selecting them. Note that although much of our methodology involves automated web scraping to build the relevant datasets, throughout our experiments we frequently manually reviewed the data collected to ensure its quality and completion.

3.1.1 Cryptographic Software vs. Cryptographic Vulnerabilities

We pause here to draw an important distinction between cryptographic software and cryptographic vulnerabilities: cryptographic vulnerabilities must necessarily originate in cryptographic software [1], but cryptographic software produces a broader class of vulnerabilities that are not only cryptographic in nature. Heartbleed [7], for instance, is an infamous buffer overflow vulnerability that originated in OpenSSL, the most widely used cryptographic library. Lazar et al. [51] conducted a methodologically similar study of 269 cryptographic vulnerabilities, finding that only 17% of the vulnerabilities they studied originated in the source code of cryptographic libraries, with the majority coming from improper uses of the libraries. Because they studied only cryptographic vulnerabilities, they excluded a significant percentage of vulnerabilities present in cryptographic software, including Heartbleed.

3.1.2 Vulnerability Data Sources

We use the National Vulnerability Database (NVD) [98], managed by the National Institute of Standards and Technology (NIST), as a standardized source of vulnerabilities from which to construct our dataset. When a new CVE (Common Vulnerabilities and Exposures) ID [63] is created, the NVD calculates a severity score (CVSS) [97] and performs additional analysis before adding the vulnerability to the NVD. While individual product bug trackers often provide more granularity, they do not allow us to standardize or compare across systems as the NVD does.

We scrape CVE data from two third-party platforms, CVE Details [31] and OpenCVE [73], which contain much the same data as the official NVD but orga-

[1]There are exceptions to this statement when the vulnerability derives from an *absence* of cryptography (i.e. data that should have been encrypted was not), but this is true as a general rule.

nize CVEs by product and vendor, enabling us to retrieve all CVEs for a particular system. OpenCVE is a newer replacement for CVE Details, which has not been updated since mid-2019. We began our study prior to the release of OpenCVE in December 2020, and so we use data from both CVE Details and Open CVE.

In total, our dataset consists of $n = 305$ CVEs in cryptographic libraries and 2,000+ CVEs in non-cryptographic software. In our study of cryptographic vulnerability characteristics, we consider only CVEs published by the NVD between 2010 and 2020, inclusive.

3.1.3 CVE Reporting Practices

Since our analysis relies heavily on the quality of the vulnerability reporting in the National Vulnerability Database, we first describe observed reporting practices in both cryptographic and non-cryptographic systems.

How reliable is the CVE data on cryptographic libraries?

After manually reviewing CVEs reported in approximately ten of the most widely used cryptographic libraries, we observe that OpenSSL has a far greater number of CVEs than any other cryptographic library, with 153 CVEs published during our timeframe of 2010 - 2020 compared to the second-highest count of 43 CVEs in GnuTLS. The question, then, is whether this difference is due to variations across libraries in security, attention, internal reporting policies, or some linear combination of the three.

We examine the CVE data on the LibreSSL and BoringSSL forks of OpenSSL as a test case to study how the official NVD vulnerability counts compare. While OpenSSL's absolute vulnerability count from 2010 - 2020 was 153 CVEs, LibreSSL and BoringSSL recorded just 7 and 4 CVEs, respectively, across the same time period. Since we conducted extensive manual examination of project commits, team mailing lists, and security advisories as part of our case study of the forks (described in greater detail in Section 4.3), we confirm that the NVD count of vulnerabilities affecting LibreSSL and BoringSSL is substantially lower than the actual number of vulnerabilities affecting those libraries. While many of these vulnerabilities may have been originally introduced into OpenSSL, because they also affected LibreSSL and BoringSSL through residual OpenSSL code they should have been included in the count under CVE Numbering Authority (CNA) standards [62]. We discuss this discrepancy further in Section 4.3.1. This finding reinforces the unreliability of absolute vulnerability counts as an indication of project security.

How do CVE reporting practices vary between cryptographic and non-cryptographic systems?

Overall, CVE reporting quality is more consistent and accurate among cryptographic software and systems focused on security (such as Wireshark and Tor). Of the top 10 software products with the highest number of reported CVEs as calculated by Li et al. [55], after manual review we find that only 3 of them—Ubuntu, Wireshark, and Android—consistently report affected system versions. Most projects report only the version in which the CVE was patched.

Non-cryptographic projects also report a skewed distribution of CVEs more frequently, as evidenced by vulnerability severity score (CVSS). For instance, we find that Android's average and median CVSS score for CVEs published from 2012 through 2020 were 6.88 and 7.2, respectively, suggesting internal practices that favor reporting only higher-severity vulnerabilities. By contrast, median severity in cryptographic libraries is very consistently 5.0, as discussed in the subsequent analysis sections.

In summary, we observe the following common issues with CVE reporting across cryptographic and non-cryptographic systems:

1. The number of CVEs reported in a project is dubiously small relative to how widely used the project is. We expected this of non-cryptographic systems, but found this is also true of a suprising number of cryptographic libraries. For instance, only 3 CVEs have ever been reported in cryptlib [30], a commonly used library.

2. CVEs reported skew towards high-severity vulnerabilities, suggesting vulnera-

bilities reported as CVEs are being somewhat cherry-picked by the vendor.

3. Affected versions are frequently not reported, making it difficult to find the version in which a CVE was introduced.

3.1.4 Systems Analyzed

We select cryptographic and non-cryptographic systems for inclusion in our study on the basis of the following requirements:

1. **Open-Source:** A critical component of this work is measuring software characteristics of codebases at particular points in time, and so we consider only systems where we have access to the source code.

2. **Written in C/C++:** To ensure an accurate comparison, all systems we select must be primarily written in C or C++. For instance, we exclude Bouncy Castle from our list of cryptographic libraries because its primary distribution is written in Java [52]. Prior work [40] has demonstrated significant differences in vulnerability causes in memory-unsafe C/C++ source code compared to systems written in memory-safe languages such as Java.

3. **Sufficient CVE Reporting:** We further select for systems that have sufficient quantities of CVEs reported to allow us to generalize. In cryptographic software, we include only cryptographic libraries that have at least 10 CVEs published from 2010 - 2020. In non-cryptographic software, to ensure that we avoid systems which are under-reporting CVEs, we select only non-cryptographic systems with comparatively high raw vulnerability counts relative to all products in the NVD. Both non-cryptographic systems we select are among the top 10 software products with the largest number of CVEs as studied by Li et al. [55].

In line with the criteria above, we study the following six cryptographic libraries: OpenSSL [74], GnuTLS [41], Mozilla Network Security Services (NSS) [65], Botan

	Cryptographic Library	Num. of CVEs
1.	OpenSSL	153
2.	GnuTLS	43
3.	Mozilla NSS	40
4.	WolfSSL	36
5.	Botan	21
6.	Libgcrypt	12
	Total	**305**

Table 3.1: The six cryptographic libraries studied, listed in order of the total CVEs published in each library between 2010 and 2020, inclusive.

[23], Libgcrypt [56], and WolfSSL [102]. We additionally collect data from the LibreSSL [57] and BoringSSL [21] libraries, though they do not meet our CVE reporting requirement, as part of a case study of OpenSSL discussed further in Section 4.3.

We also study two non-cryptographic systems, Ubuntu Linux [95] and Wireshark [101], as part of our comparison of the security impact of complexity (discussed in Chapter 5). We are limited in the number of non-cryptographic systems we can consider by the quality of the CVE data available. In order to accurately track vulnerabilities within a system, we need software systems that report all product versions affected for each CVE, not merely the patched version. This level of granularity gives us the version in which a CVE was introduced, which we use to calculate both exploitable lifetime and individual version security. As we discuss further in Section 5.2, very few non-cryptographic systems consistently report such CVE version data.

3.2 Vulnerability Type

Vulnerabilities in cryptographic software can be divided into several different categories based on error type. The National Vulnerability Database assigns each vulnerability a Common Weakness Enumeration (CWE) [64] indicating the official vulnerability type classification. Of the 305 CVEs in our dataset, 37 do not have a CWE listed, so we exclude them in this section for a new total of $n = 268$ CVEs. Because CWE labeling can be inconsistent and overly granular, particularly when

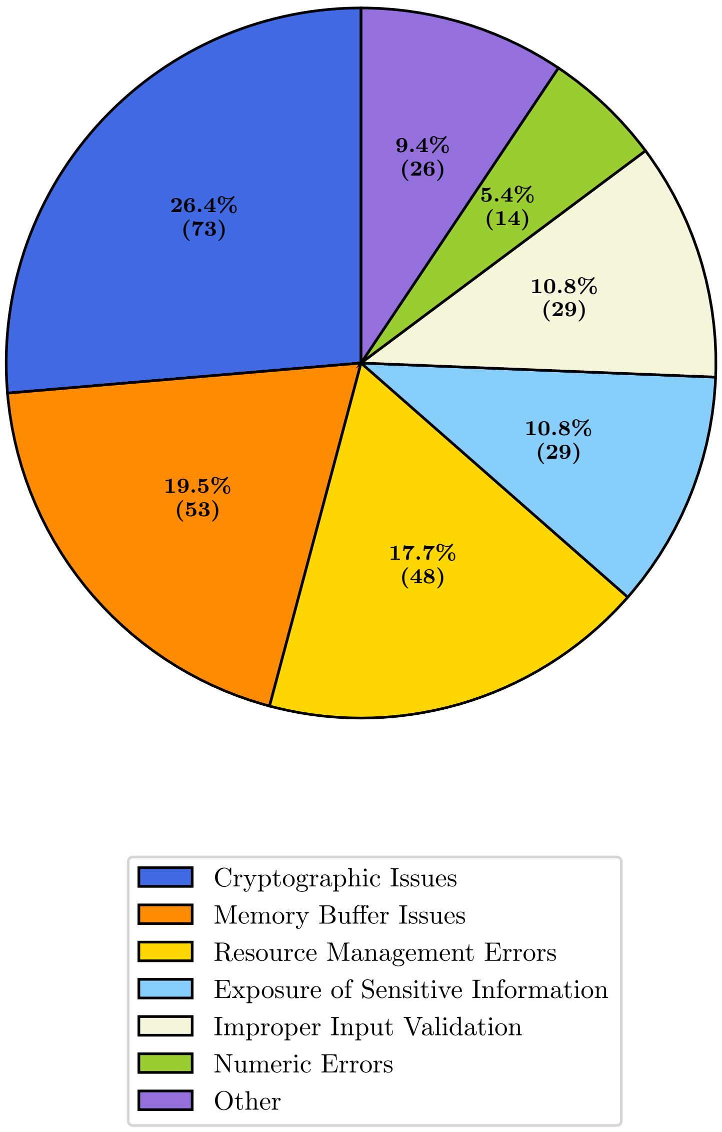

Figure 3-1: Percentage breakdowns of vulnerability types according to CWE description.

	System	Num. of CVEs with CWEs	Num. of Cryptographic CVEs	% Cryptographic
1.	OpenSSL	134	38	28.35%
2.	GnuTLS	39	12	30.76%
3.	Botan	19	2	10.52%
4.	Libgcrypt	12	3	25%
5.	WolfSSL	32	9	28.13%
6.	Mozilla NSS	32	9	28.13%
	Total	**268**	**73**	**27.24%**

Table 3.2: Percentage breakdown of cryptographic versus non-cryptographic CVEs in cryptographic libraries.

studied across multiple systems and years, we further manually group CVEs into a smaller number of categories. For instance, CWE-125: Out-of-bounds Read, CWE-787: Out-of-bounds Write, and CWE-131: Incorrect Calculation of Buffer Size can all be grouped under the larger category of Memory Buffer Issues. The 305 vulnerabilities we studied had 36 distinct CWEs, so we combined these 36 CWEs into 7 more general categories.

Figure 3-1 shows each of the 7 categories created, the absolute number of CVEs in each category, and the relative percentages of the total. While cryptographic issues are the largest individual category, comprising 26.4% of CWEs, memory-related errors are the most common overall type, producing 37.2% of CWEs when combining memory buffer issues and resource management errors. A further 21.3% of CWEs arise from various smaller sub-categories, including exposure of sensitive information, improper input validation, and numeric errors.

3.2.1 Cryptographic Vulnerabilities

How many vulnerabilities in cryptographic libraries are actually cryptographic? We previously distinguished between cryptographic software and cryptographic vulnerabilities in Section 3.1.1. Cryptographic software can produce plenty of vulnerabilities where the error is non-cryptographic, as shown in Figure 3-1.

We define "cryptographic vulnerability" more broadly than Lazar et al. did in their study of cryptographic CVEs. They considered only CVEs tagged as "CWE-310: Cryptographic Issue" under the NVD categorization, which is the largest cryptographic CWE category. However, the NVD also contains a number of more specific CWE classifications, such as CWE-326: Inadequate Encryption Strength or CWE-327: Use of a Broken or Risky Cryptographic Algorithm, that should also broadly be considered cryptographic issues. We find six such CWE categories [2] (representing 25 CVEs) and include CVEs classified under them in our accounting of cryptographic issues. For CVEs with multiple CWEs, if at least one of the CWEs is cryptographic then we consider the CVE to also be cryptographic.

Our findings show that just 27.24%, or approximately 1 in 4, CVEs in cryptographic software are actually cryptographic. Table 3.2 shows the breakdown of cryptographic CVEs in each of the six cryptographic libraries we study. The percentages are remarkably consistent across libraries, with the exception of Botan where only 10.52% of CVEs are cryptographic.

High-Severity Cryptographic Vulnerabilities

This section gives additional focus to high-severity vulnerabilities since these vulnerabilities are risky enough that they almost by definition introduced an unacceptable amount of risk into the system in which they were discovered. The severity definition used here is one defined by the National Vulnerability Database, which assigns a severity rating from 1.0 to 10.0 for all CVEs in its database.

Of high-severity vulnerabilities, how many are cryptographic? We also investigate the prevalence of cryptographic CVEs in high-severity vulnerabilities specifically. CVSS v2.0 defines vulnerabilities with a score of 7.0 - 10.0 to be "high" in severity [71], so we exclude CVEs not in that range. We find that 62 of the 291 CVEs we study are classified as high-severity. Of these 62 CVEs, 7 had no CWE listed, 2 had a cryptographic CWE, and 53 were non-cryptographic. Of the most severe CVEs, just 3.6% were cryptographic, a substantially lower percentage compared to

[2]We consider the following CWEs to be cryptographic: CWE-310, CWE-327, CWE-326, CWE-320, CWE-335, CWE-330, CWE-311.

27.24% of all CVEs.

Since we observed a significant difference in cryptographic vulnerability frequency in severe vulnerabilities compared to the total vulnerability population under CVSS 2.0 scoring, we also examine high-severity vulnerabilities under CVSS 3.0, the most recently released CVSS scoring system. CVSS v3.0 further divides severe vulnerabilities into two sub-categories: “high” for scores in the range of 7.0 - 8.9, and “critical” for scores between 9.0 - 10.0. Due to qualitative changes in the scoring system from v2.0 to v3.0, there is a much larger number of severe vulnerabilities: 104 CVEs are either “high” or “critical.” Of these 104 CVEs, 8 had no CWE listed, 85 had a non-cryptographic CWE, and 11 had a cryptographic CWE. We calculate that 10.57% of severe vulnerabilities are cryptographic.

Overall, while roughly 25% of all vulnerabilities in cryptographic libraries are cryptographic errors, only 3.6% to 10.57% of the most severe vulnerabilities are cryptographic.

3.3 Vulnerability Lifetime

Determining how long a vulnerable was actively exploitable, rather than how long it existed in the source code, is critical to fully understanding its security impact. We therefore define a vulnerability’s lifetime as the period of time in which it can be exploited by a malicious actor.

3.3.1 Calculating Vulnerability Lifetime

A vulnerability is exploitable from the moment the source code in which it lives is released until the patch for the vulnerability is installed by an affected client. Calculating this lifetime, then, involves determining the release date of the first version affected and the release date of the patch. Because clients frequently continue using outdated versions without updating, even in the wake of a major security breach [37], these calculations represent a lower bound on the actual exploitable lifetime.

We calculate a vulnerability’s “exploitable” lifetime by measuring time elapsed between the version in which the vulnerability was introduced and the version where the vulnerability was patched. Of the eight cryptographic libraries considered in this study, we find that only three of them (OpenSSL, GnuTLS, and Mozilla NSS) consistently report data on versions affected for a CVE, and so we calculate lifetimes for these systems only. Within these three libraries, we further remove CVEs that did not report affected versions or did not do so accurately, giving us a final sample size of 198 vulnerabilities in total.

Determining Affected Versions

The NVD's reporting requirements ask CVEs to indicate either the patch version or the full list of affected versions. Because the patch version is usually the latest version of the system, it is much easier to obtain and report, and so the vast majority of CVE listings contain only the patch version.

Our first step, then, is to manually review products and the CVEs reported in them to ensure we only collect data from systems that consistently and accurately record affected versions. For each CVE ID, CVE Details provides a list of versions affected in a tabular format. In our review, we encounter several common types of errors with reporting of affected versions:

1. **No Versions Listed:** A CVE had only "-", ".", or a blank space listed as the affected version, indicating that no versions related to the CVE are known, as in CVE-2018-16868 in GnuTLS [33]. We necessarily exclude these vulnerabilities from lifetime calculations.

2. **One Version Listed:** A significant percentage of CVEs had only one affected version listed. After manually reviewing several hundred CVEs and patches, we conclude that this is almost always in error and represents a situation where only the current version (as of CVE discovery) is listed instead of all affected versions. For example, the text description of CVE-2017-7869 in GnuTLS [32] indicates the vulnerability is fixed in 3.5.10, and the version affected is given as 3.5.9. A manual review of the references reveals that the CVE was in fact

introduced well before version 3.5.9. We eliminate such vulnerabilities from our dataset, since to include them would artificially lower the lifetime data collected.

3. **All Versions Listed:** In some systems, the text description of the CVEs gives the affected versions as some variation of "version X and before", and the table containing affected versions lists all versions the system has ever released. Chrome, for instance, lists every single one of 4,742 versions as affected versions for all CVEs published in 2019 because the description text includes the phrasing "prior to version [X]" [34]. While the table produced is a technically correct reading of the CVE description, we found that this reporting is almost always incorrect and should not be interpreted literally. Our approach here differs from Rescorla [85], who similarly measured vulnerability lifetime as exploitable lifetime and collected affected version data but interpreted phrasing like "version X and earlier" at face value. In our work, we avoid studying any systems that consistently exhibit this version reporting pattern.

After thorough manual review, we identified three cryptographic systems and two non-cryptographic systems that have sufficiently accurate version reporting, as described in Section 3.1.4. For each of these systems, we use Python's BeautifulSoup library [15] to scrape all affected versions from CVE Details, sorting the versions alphanumerically to obtain the first affected version and the patch version. In the case of OpenSSL, the project itself keeps a detailed record of vulnerabilities and affected versions, so we scrape from the project's vulnerabilities page [6] instead.

Mapping Versions to Release Dates

To use the initial and patch versions of a CVE to calculate the CVE's lifetime, we need to know the release dates of those versions. For each system we study, we construct a dataset of versions and release dates through manually scraping individual system websites and developer mailing lists for version release dates. While most systems clearly publish the release dates of major versions, we found that minor version release dates were trickier to track down, particularly for versions released over a decade ago,

	System	**Num. CVEs**	**Median Lifetime**	**Avg. Lifetime**	**StdDev Lifetime**
1.	OpenSSL	153	1511	1631.92	1163.65
2.	GnuTLS	21	565	857.09	832.58
3.	Mozilla NSS	24	2625	2865	2565.32
	Total	**198**	**1448**	**1669.6**	**1483.26**

Table 3.3: Exploitable lifetimes (in days) of vulnerabilities in cryptographic libraries.

and required substantial manual trawling of various mailing lists. In a small number ($n = 3$) of cases, we were only able to find the month and year of a release date, and not the precise date. In these cases, we used the 15th of the month as an approximation of the release date. We then use this dataset to efficiently look up the dates of the initial and patch versions for a CVE.

Given the time-consuming nature of determining release dates for all versions of a system and the difficulty of finding systems that accurately report affected versions for each vulnerability, we limited ourselves to collecting CVE lifetime data from three cryptographic libraries (OpenSSL, GnuTLS, and Mozilla NSS) and two non-cryptographic systems (Ubuntu and Wireshark).

3.3.2 Exploitable Lifetimes

Table 3.3 displays the median and average lifetimes for each system along with the sample standard deviation. We find that overall, the median lifetime of a vulnerability in cryptographic software is 1,448 days, or 3.97 years. In all three systems, the average lifetime is slightly greater than the median due to a small number of outlier CVEs that persist in the software for abnormally long periods. Because these calculations necessarily include only vulnerabilities that have been discovered and reported, these numbers should be interpreted as lower bounds on the actual lifetimes.

3.4 Vulnerability Severity

We further study vulnerability severity across systems using CVSS v2 scores. While the latest CVSS v3 is the most recent CVSS version, the NVD's policy was not to retroactively score vulnerabilities published prior to December 20, 2015 according to the CVSS v3.0 scale [69], so a number of vulnerabilities in our dataset only have v2 scoring. To maintain consistency, we use CVSS v2 for all CVEs.

Across the six cryptographic libraries studied (OpenSSL, GnuTLS, Botan, Libgcrypt, WolfSSL, NSS) and the 305 CVEs affecting them, the average severity score was 5.21, with a standard deviation of 1.73. All libraries except Libgcrypt had a median severity of 5.0, with Libgcrypt's median severity slightly lower at 4.3. There was very little variation among libraries in vulnerability severity. We conclude that an average severity score of around 5 is a sign of healthy CVE reporting and scoring. For a severity score to be higher would suggest that the system is under-reporting CVEs by reporting only the more severe vulnerabilities.

Having now surveyed the characteristics of the vulnerabilities themselves independent of the software implementations in which they originated, in the upcoming chapter we further investigate the causes of these vulnerabilities within cryptographic software.

Chapter 4

Cryptographic Complexity and Security

Excessive complexity is often cited within the security community as the reason for adverse security outcomes. We previously discussed how the idea that complexity is the enemy of security has become a common refrain within the security community. However, the field lacks empirical evidence supporting that truism with respect to cryptographic software. In this chapter, we provide such evidence through an analysis of the relationship between complexity and vulnerability count in cryptographic libraries with sufficient CVE reporting, using codebase size and cyclomatic complexity [59] as approximations for how complex a system is.

Despite the ever-increasing complexity of modern software products, there has been comparatively little recent work that attempts to understand the connection between increased software complexity and reported vulnerabilities. Ozment et al. [78] conducted an empirical study of security trends in the OpenBSD operating system across approximately 8 years, focusing specifically on how vulnerability lifetimes and reporting rates have changed during the relevant period. They found that vulnerabilities live in the OpenBSD codebase for over two years on average before they are discovered and patched. The researchers also found that the quantity of reported vulnerabilities decreased over the period studied, but the work is inconclusive on whether the decrease in absolute numbers is due to improved security practices or to other external factors. Zimmermann et al. [105] similarly analyzed vulnerability count in the Windows Vista operating system, finding a weak correlation between number of vulnerabilities and various common software metrics. Given how significantly vulnerability reporting practices and the software development ecosystem have changed since Ozment et al.'s experiment, it is worthwhile to ask similar questions again today, and on a larger scale.

More recently, Azad et al. [14] studied the effects of software debloating in web applications. They analyzed how removing significant percentages of the codebase in PHP web applications impacted the vulnerability count. Their analysis suggested that cryptography-related vulnerabilities are not removed through debloating, but their sample size of 3 cryptographic vulnerabilities precludes drawing meaningful conclusions. A significantly larger-scale study of cryptographic vulnerabilities will be completed as part of this thesis. Christensen et al. [28] conducted a similar vulnerability reduction study as part of an automated debloating platform they developed, but looked at firmware debloating in modern, off-the-shelf products. After pruning redundant firmware, they studied bugs per line of code (BPLOC) as a security metric. Unlike Azad et al., they did not specifically map bugs to their location in the codebase, instead estimating the number of bugs removed post-pruning. They did not differentiate between cryptographic and non-cryptographic vulnerabilities.

Since software that implements cryptographic algorithms and protocols has characteristics and testing processes that are quite distinct from non-cryptographic software, further study of the impact of complexity on security in cryptographic software specifically is needed. While these works are similar in their focus on correlations between software complexity and vulnerability count, this chapter focuses specifically on cryptographic source code, analyzing cryptography libraries and the software vulnerabilities they produced.

4.1 Methodology

We explore how the size of a codebase affects vulnerabilities introduced in two smaller studies: First, we study how many CVEs are introduced for every thousand lines of code across relevant OpenSSL versions. Second, we use the LibreSSL and BoringSSL forks of OpenSSL as a natural experiment to study how the changes made in the forks impacted the security of the projects.

4.1.1 Complexity Metrics Analyzed

There are a variety of mechanisms for approximating software complexity. We select two particular complexity metrics through which to study security outcomes across different systems: total lines of code (LOC) and cyclomatic complexity. Thomas J. McCabe first introduced the concept of cyclomatic complexity in 1976 [59], defining it as the number of linearly independent paths through a system's source code. LOC and cyclomatic complexity are complementary metrics to study: the first measures the sheer size of the codebase, while the second approximates the structural complexity of the source code. Furthermore, prior work [105, 93] has shown that these two metrics are among the best complexity predictors of vulnerabilities in non-cryptographic software, suggesting that they may also correlate well with vulnerabilities in cryptographic software.

Lines of Code: We use `cloc` [12], a command-line tool, to count the total lines of code for each language in a codebase. Throughout this thesis, we only consider C or C++ source code lines in our count, and we exclude blank lines, comment lines, and header files. We collect LOC measurements of the relevant systems at annual intervals from 2010 through 2020 as well as for specific version releases.

Cyclomatic Complexity: We use a separate command-line tool, `lizard` [104], to calculate the cyclomatic complexity of all .cpp files in a codebase. Lizard calculates the complexity of each file individually and averages them together, outputting a single net cyclomatic complexity number (CCN).

4.2 Correlation Between LOC and Vulnerability Count

In this section, we investigate how many vulnerabilities are introduced for every 1,000 lines of code introduced in cryptographic software. Comparing raw vulnerability counts, or any other metric that relies on raw vulnerability counts, across disparate systems is challenging because of variations in vulnerability reporting practices. Even software systems with almost identical purposes and similarly sized development teams can have substantially different CVE reporting policies. [1]

Instead of comparing across different systems, we instead contrast rates of CVE introduction across different versions of the same codebase: OpenSSL. OpenSSL is the only cryptographic library with both sufficient quantity and quality of CVE entries to allow us to do such a comparison with meaningful results. It is also to date still the single most widely used cryptographic library, making the findings more impactful [27, 46].

4.2.1 OpenSSL Versions

In particular, in this section we examine whether there exists a linear correlation between the lines of code introduced in a version and the number of CVEs introduced. We select four OpenSSL versions (0.9.8, 1.0.0, 1.0.1, and 1.0.2) whose release dates roughly span the 10-year period from 2005 to 2015. OpenSSL 0.9.8 was released in July 2005, and 1.0.2 was released in January 2015. Versions released within this timespan are old enough that vulnerabilities have had time to be discovered but recent enough that the results are still relevant for contemporary software development. Since OpenSSL releases major versions every two to three years on average, we are necessarily limited in the number of versions we can include in our study.

For each of the four releases, we approximate the net lines of code added in the version by measuring the overall size of the major version in question and the most recent prior version (in all four cases, a minor version) and taking the difference. The

[1]See Section 4.3, which describes CVE reporting across OpenSSL, LibreSSL, and BoringSSL, three libraries with the same foundational source code, as an example.

Major Version	Release Date	Most Recent Minor Version	LOC Change	CVEs Introduced	CVEs \ KLOC
1.0.2	1/22/2015	1.0.1l	22,236	25	1.12
1.0.1	3/14/2012	1.0.0h	18,766	33	1.76
1.0.0	3/29/2010	0.9.8n	15,510	12	.77
0.9.8	7/5/2005	0.9.7g	23,174	28	1.2

Table 4.1: CVEs introduced per thousand lines of C/C++ source code (KLOC) in four versions of OpenSSL.

source code for all versions was obtained from the releases stored in OpenSSL's source code repository [77]. A lack of data on the precise commits that were included in a version release makes it necessary to measure LOC added in a version in this indirect manner, but the LOC difference between a release and the one immediately preceding it gives a very near approximation of the size of the version.

It should be noted that this calculation yields the net change of lines added and removed, rather than solely lines added, which we feel is a better approximation for the impact of the version on the codebase. Again, in all of our measurements we consider only C & C++ source code and exclude blank lines, comment lines, and header files.

4.2.2 Rate of Vulnerability Introduction

Table 4.1 reports the rates of vulnerability introduction across the four OpenSSL versions studied. Column 4 gives the LOC change for each version, calculated as described in Section 4.2.1, for an average of 19,921.5 lines of C added per version. We further calculate the number of CVEs introduced in each version using vulnerability data tracked by the OpenSSL project [6]. To estimate the number of CVEs introduced per thousand lines of code, we then simply take the ratio of columns 4 and 5 in Table 4.1.

Column 6 shows that, on average, around 1 CVE is introduced in OpenSSL for every thousand lines of code added. This ratio should be interpreted as a lower bound since it necessarily includes only vulnerabilities that have been discovered. The ratio of vulnerabilities *existing* in the codebase per thousand LOC is very likely higher, though it is impossible to know just how much higher. Furthermore, we made a methodological decision to calculate these sizes based on the net LOC difference (which takes into account LOC removed) instead of solely considering LOC added, since this gives a more complete accounting of changes made in the version.

Comparing across versions, OpenSSL exhibits a consistent trend of larger version sizes producing higher quantities of CVEs. Spearman's non-parametric correlation test calculates a correlation coefficient of $\rho = 0.8$, indicating a strong linear relationship. The outlier to the overall trend is version 1.0.1, the version in which Heartbleed was introduced. OpenSSL's codebase received a sudden influx of attention in 2014 after Heartbleed was discovered, which may partially explain the higher number of vulnerabilities discovered in 1.0.1. If there are more researchers examining the source code, more vulnerabilities and other bugs are likely to be found.

4.3 Case Study: OpenSSL, LibreSSL, and BoringSSL

While arguably not the most serious security issue found in the OpenSSL codebase, the Heartbleed vulnerability [7] gained international attention in April 2014, bringing OpenSSL into the spotlight with it. Not only did the vulnerability compromise roughly two-thirds of Internet traffic, it was easy for an attacker to understand and exploit while difficult for servers to detect that exploitation.

The increased scrutiny of the OpenSSL codebase in the wake of Heartbleed prompted the creation of two major forks of the codebase: LibreSSL [57], developed by the OpenBSD Project and released on July 11, 2014, and BoringSSL [21], developed by Google and released on June 20, 2014. Figure 4-1 depicts this split.[2] Although LibreSSL and BoringSSL are both forks of OpenSSL, they were intended for very different purposes: LibreSSL was conceived of as a replacement for OpenSSL that maintained prior API compatibility and portability [57], while BoringSSL was originally developed for Google's internal use only. OpenBSD forked OpenSSL with the

[2]OpenSSL is itself a fork of an older library called SSLeay [94], but we omit a detailed history of OpenSSL here and simply use the first version published under the OpenSSL license in 1998.

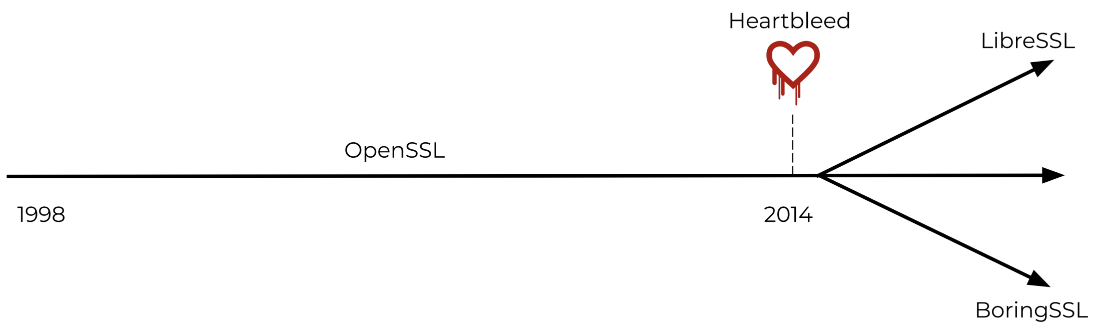

Figure 4-1: A high-level timeline of OpenSSL and its 2014 fork in the aftermath of the Heartbleed vulnerability.

goal of creating a more modern and secure TLS library after a particular disagreement over the way OpenSSL handled memory management [57, 4]. The BoringSSL project, on the other hand, specifically states that it is not recommended for external use outside of Google [21]. This difference in purpose helps to explain why BoringSSL diverges more from OpenSSL than LibreSSL in overall size and features offered, as shown in Figure 4-2 and discussed further in the following section.

4.3.1 Code Removal

Post-fork, LibreSSL and BoringSSL both removed significant amounts of the OpenSSL codebase. On April 7, 2014, the day that Heartbleed was patched and announced, OpenSSL contained 269,179 lines of C and C++ source code. In the months that followed from early April through June 2014, LibreSSL removed roughly 60,000 C or C++ LOC while BoringSSL removed 180,000 LOC.

Figure 4-2 shows a comparison of the codebase sizes over time, beginning in July 2014 once all three libraries had been released. The comparatively large size of OpenSSL relative to the other two is primarily due to OpenSSL's maintenance of legacy ciphers and protocols in order to maintain backwards compatibility and portability. Additionally, the abrupt spike in size of OpenSSL's codebase in 2018 was due to temporary testing files added as part of NIST's Cryptographic Algorithm Valida-

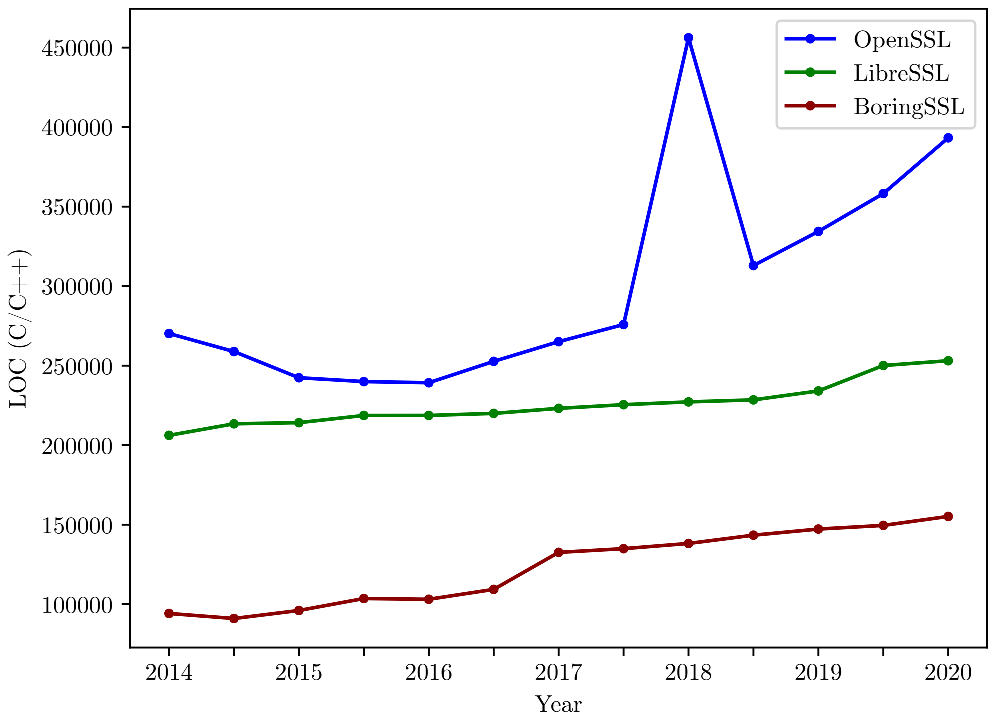

Figure 4-2: Relative sizes of OpenSSL, LibreSSL, and BoringSSL over seven years from July 2014 through July 2020. LOC count includes only C and C++ source code and excludes blank lines, comment lines, and header files. The unusual 2018 spike in OpenSSL's repository size is due to temporarily added testing files for FIPS 140 compliance.

tion Program [76], which is necessary for FIPS certification [70]. These files were only temporarily added to the source code repository and were never released as part of OpenSSL.

We summarize the major changes made by LibreSSL and BoringSSL in the immediate aftermath of Heartbleed below, focusing on features removed between April and July of 2014. In some cases, one or the other of LibreSSL and BoringSSL did not entirely remove a feature but refactored it extensively enough that they substantially lowered the amount of implementation code required.

LibreSSL Changes The OpenBSD team built LibreSSL under the design that the library would only be used on a POSIX-compliant operating system with a standard C compiler [96]. This assumption enabled them to remove much of OpenSSL's operating system and compiler-specific source code. The LibreSSL team further removed a handful of unnecessary or deprecated ciphers and protocols, many of which dated back to the 1990s, including SSLv2 and MD2 [3].

BoringSSL Changes BoringSSL has more limited intended use cases than LibreSSL and so was able to discard roughly three times as many LOC as LibreSSL. Since they removed anything not needed for Chromium or Android, BoringSSL discarded a variety of outdated ciphers, protocols, and other algorithms, including Blowfish, Camella, RC5, MD2, and Kerberos [50]. We also observed that BoringSSSL refactored what source code remained more extensively than LibreSSL.

4.3.2 Determining Vulnerabilities Removed

The abrupt jettisoning of 22% and 70% of OpenSSL's codebase by LibreSSL and BoringSSL, respectively, raises the question of what impact this had on the security of the two new codebases. Specifically, of the 59 vulnerabilities introduced but not yet discovered in the OpenSSL codebase as of the April 2014 fork, how many still affected LibreSSL and BoringSSL after the steps they took to shrink the codebase?

[3]From June through October 2014, the LibreSSL team chronicled their thoughts on OpenSSL and implementation changes made in LibreSSL on Twitter under the handle @ValhallaSSL.

Library	% of Codebase Removed	% of Vulnerabilities Removed
LibreSSL	22%	25% (15/59)
BoringSSL	70%	59.3% (24/59)

Table 4.2: Percentages of source code and CVEs removed in LibreSSL and BoringSSL compared to OpenSSL.

To answer this question, we study vulnerabilities reported in OpenSSL in the wake of Heartbleed and map each vulnerability to its locations in LibreSSL and BoringSSL.

Since we find the official National Vulnerability Database count wholly inaccurate for tracking whether LibreSSL and BoringSSL were affected by CVEs (as described further in Section 3.1.3), we create our own database based on OpenSSL's vulnerability list [6]. For each CVE affecting OpenSSL post-fork, we manually classify it as having also affected LibreSSL or BoringSSL as of July 11 post-code removal. To do so, we conducted an extensive manual review of commit descriptions in the source code of both libraries [58, 22], the LibreSSL team mailing list archives [3], the LibreSSL changelog [2], LibreSSL security advisories [72], and the BoringSSL bug tracker [1].

We consider only vulnerabilities introduced prior to the forks (e.g. introduced in OpenSSL version 1.0.1g or earlier) and discovered after July 11, 2014 (e.g. patched in OpenSSL version 1.0.1i or later). We select a July 11 cut-off date since that is the official release date of LibreSSL, and so by that date both BoringSSL and LibreSSL had been released. Vulnerabilities discovered and patched between April and July 11 are not included in our dataset to allow the LibreSSL and BoringSSL projects time to make initial modifications to the source code and officially release their versions of the library.

In our review of individual project resources, if the team indicated they were not affected by a particular CVE then we use the OpenSSL patch commit location to identify the relevant source code in Libre and Boring and determine why they were unaffected. If they removed the offending source code prior to July 11, then we consider the library to have been unaffected by the CVE in the context of our study. Approximately 1/3 of OpenSSL CVEs were not mentioned in any of the

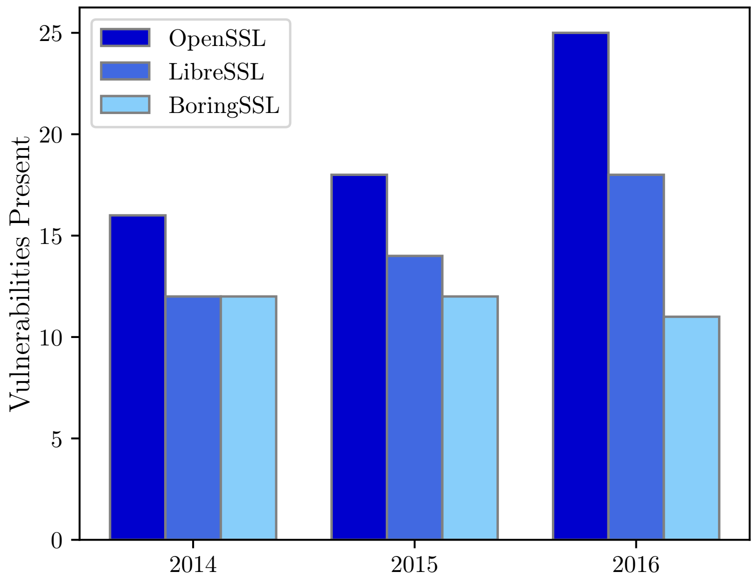

Figure 4-3: Number of vulnerabilities discovered in the codebases after the initial releases of LibreSSL and BoringSSL, organized by year of discovery in OpenSSL.

aforementioned LibreSSL or BoringSSL sources, in which case we again revert to independently reviewing the source code covered by the OpenSSL patch and use git diffs to compare files across libraries.

4.3.3 Security Impact of Code Removal

Our dataset consists of 59 CVEs introduced in OpenSSL prior to the Heartbleed fork on April 7 and discovered after the releases of both LibreSSL and BoringSSL. Table 4.2 shows that of those 59 CVEs, 44 still affected LibreSSL and just 35 affected BoringSSL. The clear correspondence between the percentage of the OpenSSL codebase removed and the percentage of OpenSSL vulnerabilities removed demonstrates the security implications for reducing codebase size.

Figure 4-3 further breaks down the total based on the year OpenSSL published the CVE. Only the years 2014 through 2016 are included in the figure because no additional CVEs were discovered in 2017 or later that were introduced prior to April

2014 (and therefore qualified for inclusion in our study).

Because these forks inadvertently created a natural experiment in reducing software complexity, we can definitively conclude that these vulnerabilities were removed from the respective codebases because the original source code was removed, a stronger conclusion than merely demonstration a correlation between size and security.

Our study of the OpenSSL forks and our separate study of vulnerability frequency in OpenSSL together allow us to provide empirical evidence of how complexity affects security. We find a lower bound of 1 vulnerability introduced for every thousand lines of cryptographic source code, and our case study of LibreSSL and BoringSSL further demonstrated a lower correspondence between source code removal and vulnerability removal. Through better understanding the causes of vulnerabilities in cryptographic software, the security community can more effectively direct their efforts in mitigating against them.

Chapter 5

Complexity in Non-Cryptographic Software

The findings detailed in Chapters 3 and 4 raise the follow-up question of whether these findings are unique to cryptographic software. In this chapter, we study how the relationship between complexity and security in cryptographic systems compares to non-cryptographic systems. Unlike in Chapter 3, we do not study vulnerability type or severity in non-cryptographic systems since there has been an extensive body of work on that subject already [55, 79]. Rather, we apply the same methodology used in Chapter 4 to measure complexity in various non-cryptographic systems.

5.1 Cyclomatic Complexity Comparison

As a proxy for estimating the overall complexity of a codebase, we use McCabe's cyclomatic complexity measure [59], defined as the number of linearly independent paths through the source code. To vary the systems we selected, we studied an operating system (Ubuntu Linux), a web browser (Chrome/Chromium), and a software application with a security focus (Wireshark).

Table 5.1 shows the relative cyclomatic complexities for the three main cryptographic libraries studied and the three non-cryptographic systems selected. We measured only the cyclomatic complexity of the most recent version of the system as of

	System	Version	Average CCN
Crypto	OpenSSL	1.1.1i	6
	GnuTLS	3.7.0	5.5
	NSS	3.60.1	4.5
Non-Crypto	Ubuntu Linux	20.10	4.3
	Wireshark	3.4.2	2.9
	Chromium	89.0.4375.0	2.1

Table 5.1: Average cyclomatic complexity of C/C++ files in cryptographic and non-cryptographic systems as of December 2020.

December 2020 because unlike other metrics such as lines of source code, cyclomatic complexity rarely changes substantially across versions after the initial codebase has already been established.

The three cryptographic libraries have CCNs of 6, 5.5, and 4.5, all of which are higher than the CCNs of the three non-cryptographic systems. OpenSSL, in fact, used to have an even higher cyclomatic complexity number of 6.9 as measured in the wake of Heartbleed. The CCN dropped only after extensive improvements to codebase quality. While this finding does not directly demonstrate subsequent security consequences, it empirically demonstrates the common wisdom that the source code of cryptographic software is fundamentally structured differently and more complexly than source code in non-cryptographic systems.

5.2 Rate of CVE Introduction

To directly study the impact of complexity on vulnerabilities in non-cryptographic systems, we again use a ratio of CVEs reported relative to the number of lines of code introduced. Vulnerability count is too subjective to be used as an absolute metric across systems that vary so widely, and so to more fairly compare security we control for codebase size.

Table 5.2 reports the number of CVEs per thousand lines of C and C++ source code across two cryptographic systems and two non-cryptographic systems. In selecting systems, we are limited by the quality of the data available. As discussed in

	System	Versions	LOC Change	CVES Introduced	CVEs / KLOC
Crypto	OpenSSL	1.0.0 - 1.0.2	58,980	70	1.19
	NSS	3.12 - 3.21	35,554	30	0.84
Non-Crypto	Ubuntu Linux	10.10 - 15.10	3,629,869	1,313	0.36
	Wireshark	1.4.0 - 2.0.0	994,607	301	0.3

Table 5.2: CVEs per thousand lines of code in cryptographic and non-cryptographic systems.

the methodology, we can only study projects that consistently and accurately report affected versions of vulnerabilities, and so we no longer include Chromium since manual inspection of CVEs reported showed that Chromium, like many systems, reports only the version in which a CVE was patched. We also exclude GnuTLS for similar reasons.

We further filtered systems by CVSS score on the basis that a system with high-quality vulnerability reporting practices would have an average and median CVSS score of around 5.0. For instance, we originally included Android in our study but later excluded it after finding that its vulnerabilities had unusually high CVSS scores, as discussed in Section 3.1.3, indicating a strong skew towards reporting only high-severity vulnerabilities.

Of the systems studied, we collect the source code repository LOC difference during the six-year period of approximately 2010 through 2015, using 2015 as the cutoff year to allow time for CVEs to be discovered and reported. We further calculate the number of CVEs introduced during that time period, necessarily including only CVEs that accurately listed versions affected. As Table 5.2 shows, OpenSSL and Mozilla NSS, both cryptographic libraries, have substantially higher rates of CVEs per source code introduced. The other two non-cryptographic systems have vulnerability introduction rates of roughly 1/3 those of OpenSSL and NSS.

5.3 Exploitable Lifetime

The actively exploitable lifetime of a vulnerability helps us further understand security outcomes in software. We collect lifetime data from Ubuntu and Wireshark on CVEs published between 2010 and 2020, inclusive, to compare with the vulnerability lifetimes in cryptographic libraries found in Section 4.2.3.

For Ubuntu, of the 1,732 CVEs studied the average and median lifetimes are 3.91 and 4.02 years, with a standard deviation of 1.37 years. For the 401 CVEs in Wireshark, the average and median lifetimes are 1.17 and 1.30 years. We find no comparable difference compared to cryptographic systems and conclude that vulnerability lifetime is mainly a function of a particular project's version release patterns and patching policies and tells us little that is generalizable across systems.

5.4 Discussion

In our study of non-cryptographic systems, we find that code complexity in cryptographic software is substantially greater than in non-cryptographic software across all systems studied, and that non-cryptographic source code generally has a lower density of CVEs introduced compared to cryptographic libraries. Our findings suggest that cryptographic source code is indeed more brittle and prone to producing security bugs than a comparable amount of source code in a web browser or operating system. The empirical data leads us to conclude that complexity is an even worse enemy of security in cryptographic software than in non-cryptographic software.

Chapter 6

Technical Proposals for Exceptional Access

In this chapter, we systematize a number of technical proposals for implementing exceptional access, summarizing the schemes and objections raised against them. These proposals are not necessarily intended as arguments in favor of exceptional access, but were put forward to provide concrete ideas for technical discussion. Since law enforcement can already obtain unencrypted user metadata through the usual warrant process, these proposals focus on obtaining the actual content from the devices or encrypted messaging services.

We focus on contemporary proposals (introduced around 2016 or later)[1] since Denning and Branstad [36] previously presented an extensive taxonomy of the key escrow systems of the 90s and Abelson et al. [13] summarized the risks inherent in those schemes in 1997. For reasons of perceived technical feasibility, most contemporary proposals have focused on providing encrypted device access, not access to communications data.

The majority of technical proposals presented and discussed in this section incorporate the idea of key escrow in their design, in which some trusted authority is charged with storing the decryption key(s). Proposals put forth in the 90s-era encryp-

[1]We also include a discussion of the 1993 Clipper Chip system since it is both the most well-known scheme for lawful access and the only one put forward by the United States government.

tion debate were almost exclusively based on the concept of key escrow [61, 60, 16], and many of the contemporary proposals are modified versions of previously suggested mechanisms for key escrow. Among these systems, there are variations on the identity of the trusted authority in possession of the key (device vendor or government) and how many keys are used (a single "master key" for all devices or individual keys for each device).

Finally, we note that some proposals are more detailed in their specifications than others. A lack of rigor in a proposal makes a rigorous response impossible. The more precise a proposal, the more effectively it can be evaluated by the cryptographic and security communities. The high-level summaries here, then, reflect the depth (or lack thereof) of the proposals themselves.

6.1 Proposals introducing cryptographic modifications

Based on the framework of this thesis in which we distinguish between changes to cryptographic software and changes to general-purpose software, we consider exceptional access proposals requiring cryptographic modifications separately from those that do not modify the underlying cryptography. Most historical and contemporary exceptional access schemes have proposed using cryptography to enable this access, usually through key escrow. In this section, we summarize five such schemes for device and communications access. In cases where a system has not been given a specific title, we refer to it under the name of the individual(s) proposing it.

6.1.1 Clipper Chip

We begin by discussing one of the earliest and most infamous of exceptional access proposals. In the first iteration of the encryption debate in the 90s, the U.S. government proposed a hardware-based key escrow system for voice encryption in which a "Clipper chip" would be placed in each device. Much of the design was kept secret, but in essence each device would have a two-part access key of 80 bits, where half of the access key (40 bits) was stored by the government. A Law Enforcement

Access Field (LEAF) containing the chip's ID and communication session key was encrypted using the device's 80-bit access key. Because law enforcement retained half of the access key, they would be able to break the 40-bit encryption and access the communications while anyone else would have to attempt to recover all 80 bits of the key.

The Clipper Chip was ultimately abandoned after Matt Blaze, then a research scientist at AT&T Bell Laboratories, discovered flaws in 1994 [20] that enabled users to bypass the LEAF field such that law enforcement would no longer have access to transmissions. While this exploit did not compromise the confidentiality of the communications, it was enough to force the government to abandon the idea. To date, the Clipper Chip is the only publicly available exceptional access proposal introduced by the U.S. government.

6.1.2 CLEAR

In 2017, Ray Ozzie, a former Microsoft executive, proposed a contemporary key escrow system for encrypted device access which he named "CLEAR" [81, 80, 54]. In CLEAR, each vendor will generate a public/private key pair. Each device stores the vendor's public key and generates a "vendor PIN", possibly based on the user passcode, when the passcode is first set by the user. This PIN is what will enable a third party (i.e., law enforcement) to unlock the device. The device encrypts this vendor PIN using the vendor's stored public key, and so the PIN can only be recovered using the vendor's private key, which is securely stored with the vendor.

If law enforcement has obtained a warrant granting access to a device, they contact the vendor and send them the device's encrypted PIN. The vendor uses its private key to decrypt the PIN, which it then shares with law enforcement, enabling them to unlock the device. It should be noted that accessing the PIN in the first place requires physical access to the device. Ozzie additionally proposed a mechanism for access transparency, which in this system means that once a phone has been accessed by law enforcement, it is irreversibly "bricked" such that no changes can be made to the stored data and the device can no longer be used. Overall, Ozzie's high-level idea is very similar to key escrow schemes previously proposed in the 90s in that a third-party holds the key(s) to decrypting the device using some bit of information stored on the device. The main difference is that Ozzie's proposal has the vendor holding the access keys, not the government.

One of the largest security risks inherent in this proposal and others is how to securely store the vendor's keys. Ozzie draws a comparison to signing keys used by Apple and other device vendors used to provide software updates, and argues that these vendors would be similarly capable of securely storing exceptional access keys. The primary difference lies in the frequency of use of these keys. Unlike signing keys, these exceptional access keys would likely be accessed multiple times each day, possibly by several different people. The more often the high-security vault in which Ozzie proposes the keys be stored is accessed, the more likely it is that security issues will arise, whether out of accidental incompetence or malicious intent [17, 42, 43].

6.1.3 Stefan Savage

Stefan Savage, a professor of computer science at the University of California, San Diego, also proposed a key escrow-based scheme for encrypted device access. Savage's 2018 proposal [87] bears strong similarities to the high-level idea introduced by Ozzie, but Savage's proposal is described in substantially greater technical detail and takes additional steps to mitigate security concerns posed by Ozzie's system.

Both Savage and Ozzie propose self-escrow schemes in which the device manufacturer plays an active role, thereby requiring users to place a large amount of trust in the vendor. Savage again suggests the idea of device self-escrow facilitated by the device manufacturer, where one key is stored within a device and the other with the vendor. Law enforcement would need physical access to the device to obtain the device's key, which they would then use to coordinate with the vendor to unlock the phone. Savage's scheme additionally includes some type of access transparency mechanism which would notify a user or make it clear to anyone in possession of the device that it has been unlocked by law enforcement.

Savage's proposal includes a slight twist on the standard requirement of physical possession based on time vaulting: the system would institute a time delay such that law enforcement could access the phone only once the lockup period has elapsed. Savage gives an example period of 72 hours, during which time the law enforcement agent would need to continuously send requests to the device. This requirement of *sustained* physical access further mitigates the risks of mass surveillance and covert access by law enforcement.

6.1.4 Ernie Brickell

Ernie Brickell of Brickell Cryptology, LLC, proposed a high-level sketch of a self-escrow scheme [24] which is effectively a more general version of the one proposed by Ozzie and Savage, with the added notion of a protected partition. There are additional minor distinctions: while Ozzie and Savage mandate physical access to the device, Brickell includes the possibility of remote network access in his proposal. Further, while the proposals by Ozzie, Savage, and Brickell all include some notion of device access transparency (i.e. that a user would be able to tell whether law enforcement has accessed their device), Ozzie and Savage require this transparency, while Brickell leaves open the possibility of "some instances in which a user was never informed."

Brickell's proposal is distinct from the others in that it distinguishes between the categories of data that might be stored on a phone, and proposes that law enforcement not necessarily be given access to everything on a device. Brickell gives the examples of health data (already shared with a health provider) or data confidential to an employer (and which law enforcement could retrieve from an employer as needed). To separate data which law enforcement should be given access to from data that should arguably be kept confidential, Brickell proposes constructing an access protected partition dividing device storage into two components, a main partition and a protected partition, where law enforcement's access key would only grant them access to the main partition (and not the protected partition). An "App Approval Key" would be employed to "approve" which applications can execute in the protected partition, though it is unclear what the policies for granting approval to an application might be.

6.1.5 Crypto Crumple Zones

Charles Wright and Mayank Varia, both academic computer scientists, introduced the concept of cryptographic "crumple zones" [103] as a mechanism for encrypted communications access in 2018. Wright and Varia's scheme requires law enforcement to solve two cryptographic puzzles in order to decrypt messages. Specifically, an attacker must first solve a "gatekeeper" puzzle through brute force, requiring substantial computational effort, in order to gain access to a series of less computationally involved "crumpling puzzles" within a certain time period, where each crumpling puzzle corresponds to a single message. Even the most well-resourced attackers, then, are computationally prevented from decrypting too many messages within a small time frame.

In the sense that it is designed to be breakable only through brute-force computational effort, this scheme is similar to partial key escrow proposals of the 90s [92, 16] which were designed such that law enforcement would need to recover part of each cryptographic key through brute-force in order to decrypt communications. Wright and Varia's proposal attempts to minimize the burden on users and device manufacturers; in contrast to other proposals, the responsibility for decrypting messages lies solely with the government.

The most noteworthy difference with Wright and Varia's proposal is that it would enable encrypted messages to theoretically be accessible by anyone with sufficient resources, not just law enforcement. Wright and Varia argue that the computational work required to recover the messages through brute-force would be substantial enough that only extremely well-resourced law enforcement agencies and other state actors would be able to gain access in practice, and such access would come at a high cost. While the computational effort involved may deter ordinary cyber criminals, other nefarious nation-state actors conceivably would have resources on the level required. Since the authorization in this scheme comes from the ability to solve the puzzles embedded into the keys, a foreign nation state actor could theoretically achieve the same level of access as the U.S. government.

6.2 Proposals introducing general software modifications

Not all exceptional access schemes have sought a cryptographic solution: in this section, we summarize a singular proposal for end-to-end encrypted communications access.

6.2.1 "Ghost User"

In 2018, Ian Levy and Crispin Robinson of GCHQ described in a blog post [53] a high-level proposal for law enforcement access to end-to-end encrypted messaging services. This proposed system is particularly noteworthy because it (1) was suggested by a government agency and (2) would provide access to encrypted communications (as opposed to encrypted devices, as with most other proposals)

GCHQ proposes that service providers give law enforcement officials access by silently adding them as participants to a chat group while suppressing notifications to existing participants so they would be unaware. As Levy and Robinson describe it, this proposal would not affect the end-to-end encrypted nature of the service, but would simply add an additional "end" to the conversation. This has since become colloquially known as the "ghost user" proposal due to the invisibility of the added participant. It's important to note that this proposal would not provide retroactive access: the silently added user would only have access to messages sent after they were added to the group. This substantially limits the potential scope of law enforcement's access ability.

GCHQ's proposal has been the most heavily scrutinized of any put forward in the latest recurrence of the encryption debate, in large part because it was the first concrete proposal suggested by a member of the Five Eyes intelligence alliance. The idea was not, however, generally well-received by cryptographers and technologists

[44, 49, 90, 25] primarily due to the extensive software changes it would require and the security risk those changes would introduce. While Levy and Robinson do not provide details on how they might propose implementing this, any implementation would require substantial changes to both the service provider's servers as well as to each individual client program. On the server side, the service provider would need to make modifications to add silent participants to a chat group. Since most chat messaging services, including Signal and WhatsApp, automatically send notifications to users when a new user is added to a group chat they are in, this would further require modifying the client side of the messaging service to suppress notifications. Moreover, given the client changes made, the existence of such a capability could be detected by a variety of means, including through binary reverse engineering or network traffic analysis [26].

6.3 Design improvements cannot eliminate security risk

Contemporary exceptional access proposals reflect efforts by their authors to preemptively address many of the objections raised during the encryption debate of the 90s. To mitigate the risks of mass government surveillance, most of the device access schemes discussed here include a requirement to have physical possession of the device, and most proposals for communications or device access have the device manufacturer storing the keys instead of the government. While these design advancements are welcome, any exceptional access scheme must also contend with the inevitable introduction of software bugs as part of its implementation. In the final chapter, we discuss in more detail the implications of our empirical findings in Chapters 3 through 5 for exceptional access schemes.

Chapter 7

Discussion and Conclusion

This thesis set out to better understand the connection between software complexity and security and to quantify this connection in cryptographic software. We conducted a comprehensive study of vulnerabilities originating in cryptographic software across multiple cryptographic libraries, including what their characteristics are and why they exist. Confirming a belief long held within the security community, we found evidence of a strong correlation between the complexity of these libraries and their (in)security, empirically demonstrating the potential risks of overly bloated cryptographic codebases, and further observed that excess complexity generally produces even more vulnerabilities in cryptographic libraries than in non-cryptographic software.

This final chapter discusses the main takeaways, highlighting the most noteworthy results and their implications for software development practices and the ongoing encryption debate. Returning to our earlier discussion of technical proposals for exceptional access, we reconsider these proposals in light of the results obtained in Chapters 3 through 5.

7.1 Need for a systems approach to cryptographic software

The findings of this thesis lay bare the discrepancy between the critical role cryptographic libraries hold in securing network traffic and the amount of attention paid to the software quality of the libraries. Chapter 4 demonstrated the substantial levels of software bloat in cryptographic software. At the time that Heartbleed was discovered in 2014, OpenSSL had grown to an extent that BoringSSL was able to remove approximately two-thirds of the original codebase while still providing the same core SSL/TLS functionality. Moreover, one of the more interesting trends from Figure 4-2 is that all three of OpenSSL, LibreSSL, and BoringSSL have been gradually increasing in size, to the point where as the writing of this thesis LibreSSL is roughly the same size as OpenSSL was at Hearbleed's discovery. Even projects that set out to be minimalist and security-focused, as both LibreSSL and BoringSSL did, naturally accumulate excess source code and features over time.

This excess is particularly alarming in cryptographic software given that it produces vulnerabilities at higher rates than in non-cryptographic software (as discussed in Section 5.2). Since approximately three out of every four vulnerabilities in cryptographic software are caused by common implementation errors, and particularly by memory management issues, overly bloated software threatens significant implications for library security. Furthermore, the most severe vulnerabilities are even *more* likely to have been caused by implementation errors.

All of these findings underscore that the security community needs to devote a level of attention and resources that reflect the importance of cryptographic libraries. Other studies [55, 37] investigating vulnerabilities in cryptographic or open-source software have previously called for the community to improve development and testing processes for security software. Through better understanding the underlying technical causes and properties of vulnerabilities in cryptographic software, the security community can more effectively direct their efforts in mitigating against them.

Since the scope of this thesis is to investigate the causes of vulnerabilities in software, not the causes of complexity in software, further research is needed to to understand how to more effectively support OpenSSL and other cryptographic libraries. The issue is partly an economic one: around the time Heartbleed was discovered in 2014, OpenSSL subsisted on annual donations of around $2,000 and had just one full-time developer and three "volunteer programmers" [82]. While OpenSSL's funding situation improved in the immediate aftermath of Heartbleed, as the years have worn on some of these post-Heartbleed sources of funding have not been sustained [86] and OpenSSL's current donations page is quite sparse [75]. Less well-known cryptography libraries may receive even less financial support than OpenSSL. Any comprehensive solution treating these libraries as the critical systems they are must address both the technical and non-technical causes of complexity and insecurity in order to be truly effective.

7.2 Don't roll your own crypto

Confirmation that complexity is indeed the enemy of security will not come as a surprise to most in the security community. Empirical evidence of just how *much* greater of an impact complexity has on cryptographic software, though, will hopefully help garner support for establishing secure practices and standards around cryptographic implementations. The cryptographic libraries we studied produced vulnerabilities at rates of up to one CVE for each thousand lines of code added, a rate of three times as much as in non-cryptographic software. In our analysis, we even find one instance where the patch for a vulnerability introduced a new vulnerability [5].

The findings of this thesis lend empirical support to conventional wisdom preaching the dangers of "rolling your own crypto"—meaning that software developers should always defer to established libraries and tools instead of re-implementing their own versions. As previously discussed in Chapter 3, however, cryptography libraries suffer from serious usability issues that can make it challenging for non-specialists to navigate them. A usability-centered approach to designing cryptography APIs could make it easier for developers to actually follow this maxim.

7.3 Applying cryptographic software risk to exceptional access

Here, we discuss the lessons our investigation of cryptographic software has for the ongoing encryption debate. An exceptional access capability, by definition, is an addition or modification to software or hardware. It is further implied from the definition that it would be uncommonly used (only in "exceptional" circumstances) and would likely not be examined or adjusted much once built, taking on many of the characteristics of legacy code. Specifically, the long-term consequences of general cryptographic source code maintenance provide insights on the ways in which making similar cryptographic changes when creating an exceptional access capability would impact system security.

There are two categories of technical risk that exceptional access has the potential to introduce: (1) protocol design risk and (2) system implementation risk. Much of the historical and contemporary discussion around exceptional access schemes has focused on protocol design: the flaws Matt Blaze uncovered in the Clipper chip were failures in the underlying transmission protocol, since implementation of the cipher was only shared with select vendors in tamper-resistant security modules [20]. Since there exist no publicly available implementations of exceptional access, this thesis approximates the introduction of such a capability through studying software implementations of cryptographic and non-cryptographic features.

Recent proposals put forward to implement lawful access have demonstrated an intentional shift away from modifying the underlying cryptography. Levy and Robinson of GCHQ stressed that a vendor wouldn't have to "touch the encryption" to implement their proposal [53]; Savage and Ozzie both emphasized that their proposals would not require major new cryptographic changes [87, 80]; and Wright and Varia highlighted the relative simplicity of their constructions [103]. Our findings on security risk in cryptographic software compared to general-purpose software support these design choices.

Nonetheless, while it may be *more* risky to modify cryptographic software, this should not be taken as evidence that modifying non-cryptographic software is benign. A lawful access scheme requiring extensive server and client modifications (as is the case with GCHQ's "ghost user" proposal) may still pose a substantially higher level of risk than a scheme involving minor cryptographic changes. In presenting our findings, we hope to encourage a rational and evidence-based debate around how much security risk a proposal might introduce and whether that level of risk is worth assuming to meet the needs of law enforcement.

7.3.1 Lawful Hacking

As an alternative suggestion in the encryption debate, some prominent security researchers have advocated for "lawful hacking" (i.e. taking advantage of existing vulnerabilities unknown to the device manufacturers) as an alternative to intentionally designed lawful access schemes [18]. Our findings support the technical feasibility of this approach given the outsized quantities of vulnerabilities in cryptographic software and their duration: these vulnerabilities live in the codebase 4.57 years on average, continuing to wreak havoc long after they were first introduced.

Regardless of the path forward, it is the hope of this thesis that the qualitative analysis of proposed, theoretical exceptional access systems and quantitative analysis of modern cryptographic codebases and their vulnerabilities will give technologists and policymakers a more complete and empirical understanding of the long-term security consequences of exceptional access.

7.4 Limitations

In this section, we take a moment to address a number of potential limitations of our study and describe our attempts to mitigate them.

7.4.1 National Vulnerability Database (NVD)

Reporting Bias: The NVD suffers from selection bias in which systems report vulnerabilities as they are discovered. Some vendors pay little attention to the CVE database and do not bother to register vulnerabilities as CVEs, and others skew towards only reporting high-severity CVEs. Based on our observations, this is particularly true of non-cryptographic systems.

We mitigate this through intentionally selecting only non-cryptographic systems with comparatively high vulnerability counts. We further avoid using known CVE count as an absolute metric in our analysis because of these limitations, since we find it is a better indicator of a vendor's policy than of a system's security.

Quality Bias: Even when a system has consistent and significant CVE reporting, it often does not include sufficient detail. As previously discussed, most systems do not accurately report versions affected by a CVE.

7.4.2 Systems Studied

Open-Source: All systems we study are open-source projects. It is possible that the trends we observe will not be present in proprietary, closed-source software. In order to accurately measure complexity, though, it was necessary to focus solely on open-source systems.

Human Factors: The security of a system is impacted by many human factors in addition to codebase complexity. Software development and testing practices, developer experience level, and other considerations all affect the quantity of vulnerabilities introduced but are not reflected in codebase data.

7.5 Conclusion

In this thesis, we analyzed the impact complexity has on security outcomes in modern cryptographic software, including (1) characteristics of vulnerabilities in cryptographic software and (2) correlations between various complexity metrics and corresponding vulnerability counts. We extensively characterize the vulnerabilities by lifetime, type, and severity to better understand the security impact on cryptographic software. Our findings support the common intuition that it is dangerous to maintain excess amounts of C or C++ source code, and particularly so in cryptographic software.

In some ways, the work of this thesis is as aspirational as it is conclusive. Security is an elusive concept does not lend itself easily to empirical evaluation, and various economic, human, and technical factors interact in complex ways to impact the precise level of security risk in a system. It is the hope of the author that the findings presented here provide a starting point and inspiration for future work in quantifying software risk.

Bibliography

[1] BoringSSL Bug Tracker. https://bugs.chromium.org/p/boringssl/issues/list.

[2] LibreSSL ChangeLog. https://github.com/libressl-portable/portable/blob/master/ChangeLog.

[3] LibreSSL Mailing list ARChives. https://marc.info/?l=libressl&r=1&w=2.

[4] LibreSSl with Bob Beck. https://www.youtube.com/watch?v=GnBbhXBDmwU.

[5] OpenSSL Security Advisory [26 Sep 2016]. https://www.openssl.org/news/secadv/20160926.txt.

[6] OpenSSL Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html.

[7] The Heartbleed Bug. https://heartbleed.com/.

[8] The Forum: The Future of Privacy: A Discussion with Microsoft General Counsel Brad Smith. https://www.youtube.com/watch?v=DkfunImNDrU, 2014.

[9] Feinstein says FBI paid $900,000 to hack San Bernardino shooter's iPhone. CNBC, 2017.

[10] Harold Abelson, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Michael Specter, and Daniel Weitzner. Keys under doormats: mandating insecurity by requiring government access to all data and communications. *Journal of Cybersecurity*, 1.1:69–79, 2015.

[11] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[12] AlDanial. cloc: Count Lines of Code. https://github.com/AlDanial/cloc.

[13] Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Whitfeld Diffie, John Gilmore, Peter G Neumann, Bruce Schneier, Harold Abelson, Ronald L Rivest, et al. The risks of key recovery, key escrow, and trusted third-party encryption. 1997.

[14] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: quantifying the security benefits of debloating web applications. USENIX Security Symposium, 2019.

[15] Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[16] Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 78–91, 1997.

[17] Steven M. Bellovin, Matt Blaze, Dan Boneh, Susan Landau, and Ronald L. Rivest. Analysis of the CLEAR Protocol per the National Academies' Framework. Technical Report CUCS-003-18, 2018.

[18] Steven M Bellovin, Matt Blaze, Sandy Clark, and Susan Landau. Lawful hacking: Using existing vulnerabilities for wiretapping on the internet. *Nw. J. Tech. & Intell. Prop.*, 12:1, 2014.

[19] Katie Benner. Barr Revives Encryption Debate, Calling on Tech Firms to Allow for Law Enforcement. The New York Times, 2019.

[20] Matt Blaze. Protocol failure in the escrowed encryption standard. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 59–67, 1994.

[21] BoringSSL. https://boringssl.googlesource.com/boringssl/.

[22] BoringSSL GitHub. https://github.com/google/boringssl.

[23] Botan. https://botan.randombit.net/.

[24] Ernie Brickell. A proposal for balancing access and protection requirements from law enforcement, corporations, and individuals. IACR Cryptology, 2018.

[25] Jon Callas. The Recent Ploy to Break Encryption Is An Old Idea Proven Wrong. ACLU, 2019.

[26] Nate Cardozo and Seth Schoen. Detecting Ghosts by Reverse Engineering: Who Ya Gonna Call? Lawfare Blog, 2019.

[27] Censys. https://censys.io/.

[28] Jake Christensen and Ionut Mugurel Anghel. Decaf: Automatic, adaptive, debloating and hardening of cots firmware. *USENIX Security Symposium*, 2020.

[29] National Research Council. *Cryptography's Role in Securing the Information Society.* The National Academies Press, Washington, DC, 1996.

[30] CryptLib. `https://www.cryptlib.com/`.

[31] CVE Details. `https://www.cvedetails.com/`.

[32] CVE Details. CVE-2017-7869. `https://www.cvedetails.com/cve/CVE-2017-7869`.

[33] CVE Details. CVE-2018-16868. `https://www.cvedetails.com/cve/CVE-2018-16868/`.

[34] CVE Details. CVE-2019-5840. `https://www.cvedetails.com/cve/CVE-2019-5840/`.

[35] Deciphering the Debate Over Encryption: Industry and Law Enforcement Perspectives: U.S. House Committee on Energy and Commerce, 114th Cong. (2016). (Testimony of Daniel J. Weitzner). `https://docs.house.gov/meetings/IF/IF02/20160419/104812/HHRG-114-IF02-Wstate-WeitznerD-20160419.pdf`.

[36] Dorothy E Denning and Dennis K Branstad. A taxonomy for key escrow encryption systems. *Communications of the ACM*, 39(3):34–40, 1996.

[37] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[38] Electronic Frontier Foundation. The Crypto Wars: Governments Working to Undermine Encryption. `https://www.eff.org/document/crypto-wars-governments-working-undermine-encryption`, 2020.

[39] Encryption Technology and Potential U.S. Responses: U.S. House Committee on Oversight and Government Reform, 114th Cong. (2015). (Testimony of Matthew Blaze). `https://www.govinfo.gov/content/pkg/CHRG-114hhrg25879/pdf/CHRG-114hhrg25879.pdf`.

[40] Alex Gaynor. What science can tell us about C and C++'s security. `https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/`, 2020.

[41] GnuTLS. `https://www.gnutls.org/`.

[42] Robert Graham. No, Ray Ozzie hasn't solved crypto backdoors. Errata Security Blog, 2018.

[43] Matthew Green. A few thoughts on Ray Ozzie's "Clear" Proposal. https://blog.cryptographyengineering.com/2018/04/26/a-few-thoughts-on-ray-ozzies-clear-proposal/, 2018.

[44] Matthew Green. On Ghost Users and Messaging Backdoors. https://blog.cryptographyengineering.com/2018/12/17/on-ghost-users-and-messaging-backdoors/, 2018.

[45] Matthew Green. Looking back at the Snowden revelations. https://blog.cryptographyengineering.com/2019/09/24/looking-back-at-the-snowden-revelations/, 2019.

[46] Poul-Henning Kamp. Please put openssl out of its misery: Openssl must die, for it will never get any better. *Queue*, 12(3):20–23, 2014.

[47] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[48] Susan Landau. *Listening in.* Yale University Press, 2017.

[49] Susan Landau. Exceptional Access: The Devil is in the Details. Lawfare Blog, 2018.

[50] Adam Langley. BoringSSL. https://www.imperialviolet.org/2015/10/17/boringssl.html.

[51] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[52] Legion of the Bouncy Castle. The Bouncy Castle Crypto Package For Java. https://github.com/bcgit/bc-java.

[53] Ian Levy and Crispin Robinson. Principles for a More Informed Exceptional Access Debate. Lawfare Blog, 2018.

[54] Steven Levy. Cracking the Crypto War. WIRED, 2018.

[55] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[56] Libgcrypt. https://gnupg.org/software/libgcrypt/index.html.

[57] LibreSSL. `https://www.libressl.org/`.

[58] LibreSSL GitHub. `https://github.com/libressl-portable/openbsd`.

[59] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[60] Silvio Micali. Guaranteed partial key-escrow, September 9 1997. US Patent 5,666,414.

[61] Silvio Micali. Distributed split-key cryptosystem and applications, February 15 2000. US Patent 6,026,163.

[62] MITRE. CVE Numbering Authority (CNA) Rules. `https://cve.mitre.org/cve/cna/rules.html`.

[63] MITRE Corporation. Common Vulnerabilities and Exposures. `https://cve.mitre.org/`.

[64] MITRE Corporation. CWE: Common Weakness Enumeration. `https://cwe.mitre.org/`.

[65] Mozilla Network Security Services. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS`.

[66] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.

[67] National Academies of Sciences, Engineering, and Medicine. Decrypting the Encryption Debate A Framework for Decision Makers, 2018.

[68] National Vulnerability Database. CVE-2020-13777. `https://nvd.nist.gov/vuln/detail/CVE-2020-13777/`.

[69] National Vulnerability Database. CVE FAQs. `https://nvd.nist.gov/general/FAQ-Sections/CVE-FAQs##faqLink10`.

[70] National Vulnerability Database. Security Requirements for Cryptographic Modules. `https://csrc.nist.gov/publications/detail/fips/140/2/final`.

[71] National Vulnerability Database. Vulnerability Metrics. `https://nvd.nist.gov/vuln-metrics/cvss`.

[72] OpenBSD. OpenSSL 2015-03-19 Security Advisories: LibreSSL Largely Unaffected. `https://undeadly.org/cgi?action=article&sid=20150319145126`.

[73] OpenCVE. https://www.opencve.io/.

[74] OpenSSL. https://www.openssl.org/.

[75] OpenSSL. Acknowledgements.

[76] OpenSSL GitHub. aes ctr_drbg: add cavs tests. https://github.com/openssl/openssl/commit/e613b1eff40f21cd99240f9884cd3396b0ab50f1.

[77] OpenSSL Releases. https://github.com/openssl/openssl/releases.

[78] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? USENIX Security Symposium, 2006.

[79] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, 2006.

[80] Ray Ozzie. CLEAR. https://github.com/rayozzie/clear/blob/master/clear-rozzie.pdf, 2017.

[81] Raymond Edward Ozzie. Providing Low Risk Exceptional Access, U.S. Patent 20170272248A1, Dec. 2018.

[82] Nicole Perlroth. Heartbleed Highlights a Contradiction in the Web. The New York Times, 2014.

[83] Nicole Perlroth and David E. Sanger. F.B.I. Director Repeats Call That Ability to Read Encrypted Messages Is Crucial. The New York Times, 2015.

[84] Vassilis Prevelakis and Diomidis Spinellis. The athens affair. *Ieee Spectrum*, 44(7):26–33, 2007.

[85] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[86] Rich Salz. https://twitter.com/RichSalz/status/1115011689882234882.

[87] Stefan Savage. Lawful device access without mass surveillance risk: A technical design discussion. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.

[88] Bruce Schneier. A Plea for Simplicity: You Can't Secure What You Don't Understand. Schneier on Security, 1999.

[89] Bruce Schneier. Security or Surveillance? Lawfare Blog, 2016.

[90] Bruce Schneier. Evaluating the GCHQ Exceptional Access Proposal. Lawfare Blog, 2019.

[91] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.

[92] Adi Shamir. Partial key escrow: A new approach to software key escrow. *Private communication made at Crypto*, 95, 1995.

[93] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

[94] SSLeay. `https://www.ssleay.org/`.

[95] Ubuntu Linux. `https://ubuntu.com/`.

[96] Ted Unangst. LibreSSL: More Than 30 Days Later. `https://www.openbsd.org/papers/eurobsdcon2014-libressl.html`.

[97] U.S. National Institute of Standards and Technology. CVSS Information. `https://nvd.nist.gov/cvss.cfm/`.

[98] U.S. National Institute of Standards and Technology. National Vulnerability Database. `https://nvd.nist.gov/home.cfm/`.

[99] U.S. Senate. 116th Congress, 2nd Session, 2020.

[100] James Walden. The impact of a major security event on an open source project: The case of openssl. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 409–419, 2020.

[101] Wireshark. `https://www.wireshark.org/`.

[102] WolfSSL. `https://www.wolfssl.com/`.

[103] Charles Wright and Mayank Varia. Crypto crumple zones: Enabling limited access without mass surveillance. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 288–306. IEEE, 2018.

[104] Terry Yin. Lizard. `https://github.com/terryyin/lizard`.

[105] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010.