Cryptography in the Wild: An Empirical Analysis of Vulnerabilities in Cryptographic Libraries

Jenny Blessing
jenny.blessing@cl.cam.ac.uk
University of Cambridge
Cambridge, UK

Michael A. Specter
specter@gatech.edu
Georgia Institute of Technology
Georgia, USA

Daniel J. Weitzner
weitzner@mit.edu
MIT
Massachusetts, USA

ABSTRACT

The security of the Internet and numerous other applications rests on a small number of open-source cryptographic libraries: A vulnerability in any one of them threatens to compromise a significant percentage of web traffic. Despite this potential for security impact, the characteristics and causes of vulnerabilities in cryptographic software are not well understood. In this work, we conduct the first *systematic*, longitudinal analysis of cryptographic libraries and the vulnerabilities they produce. We collect data from the National Vulnerability Database, individual project repositories and mailing lists, and other relevant sources for all widely used cryptographic libraries.

In our investigation of the causes of these vulnerabilities, we find evidence of a correlation between the complexity of these libraries and their (in)security, empirically demonstrating the potential risks of bloated cryptographic codebases. Among our most interesting findings is that 48.4% of vulnerabilities in libraries written in C and C++ are either primarily caused or exacerbated by memory safety issues, indicating that systems-level bugs are a major contributor to security issues in these systems. Cryptographic design and implementation issues make up 27.5% of vulnerabilities across all libraries, with side-channel attacks providing a further 19.4%. We find substantial variation among core library components in both complexity levels and vulnerabilities produced: for instance, over one-third of vulnerabilities are located in implementations of the SSL/TLS protocols, providing actionable evidence for codebase quality and security improvements in these libraries.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Cryptography, Vulnerabilities, Complexity, Cryptography Libraries

ACM Reference Format:
Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. 2024. Cryptography in the Wild: An Empirical Analysis of Vulnerabilities in Cryptographic Libraries. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3634737.3657012

1 INTRODUCTION

Cryptographic libraries are responsible for securing virtually all network communication yet have produced notoriously severe vulnerabilities. In 2014, OpenSSL's Heartbleed vulnerability [1] enabled attackers to read the contents of servers' private memory. More recently, in June 2020 GnuTLS suffered a significant vulnerability allowing a remote attacker to passively decrypt network traffic [5]. Given the critical role these libraries play, a single vulnerability can have tremendous security impact—at the time of Heartbleed's disclosure, up to 66% of all websites were vulnerable [1].

A common aphorism in applied cryptography is that cryptographic code is inherently difficult to secure due to its complexity; that one should not "roll your own crypto." In particular, the maxim that complexity is the enemy of security is a common refrain within the security community. Since the phrase was first popularized in 1999 [88], it has been invoked in general discussions about software security [64] and cited repeatedly as part of the debate surrounding the additional complexity requirements in government encryption mandates [57]. Conventional wisdom holds that the greater the number of features in a system, the greater the risk that these features and their interactions with other components contain vulnerabilities.

Intuitively, there are a number of reasons that cryptographic code *should* suffer from vulnerabilities not seen in other systems. Timing attacks, newfound breaks in the cryptographic algorithms and protocols used, use of insecure sources of randomness, and other subtle issues appear to be relatively unique to software in this domain [59, 71]. Worse, the need to handle such unique categories of attacks may result in code that is more convoluted and difficult to decipher, leading to further software complexity, fewer reviews or analysis, and, ultimately, bugs.

Unfortunately, there is a lack of empirical evidence demonstrating the collective security impact of these cryptographic-style issues in comparison to standard systems problems. Indeed, a nontrivial portion of many common cryptographic libraries includes code that is not purely cryptographic in nature, including often complex network protocols, data serialisation (e.g., X.509 parsing), and system configuration code. This analysis is increasingly important as the industry moves toward newer and more novel cryptographic primitives, programming languages, and use-cases such as cryptocurrencies and zero-knowledge proofs. An oft-cited strategy to improve security in cryptographic libraries and other systems is to write code in memory-safe languages such as Rust, but we cannot

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3657012

evaluate the utility of such proposals without a systematic understanding of the causes of security issues in the first place [68, 70, 91].

In this paper, we conduct the first comprehensive, longitudinal analysis of cryptographic libraries and vulnerabilities affecting them. We examine 37 commonly used cryptographic libraries (see Appendix A for the full list) and include in this study the 23 libraries that have had any vulnerabilities reported in the National Vulnerability Database (NVD) [81] across the 18-year period from 2005 through 2022. In our analysis, we combine data from the NVD with information collected from project repositories, internal mailing lists, project bug trackers, and other external references. We extensively characterize the vulnerabilities originating in cryptographic software, categorizing them by root cause and feature location within the codebase. We additionally measure exploitable lifetime and severity to better understand their security impact.

Having surveyed the characteristics of the vulnerabilities themselves, we further investigate the causes of these vulnerabilities within cryptographic software. We pay particular attention to the relationship between software complexity and vulnerability frequency, recording codebase size and cyclomatic complexity [77] of each library. We also conduct an in-depth case study of OpenSSL and its major post-Heartbleed forks, LibreSSL and BoringSSL, examining how the codebases diverged and quantifying the security impact of the changes made.

Among our findings are that while 27.5% of vulnerabilities in cryptographic software are issues directly related to the cryptographic protocols or implementation, 40.0% of errors across all libraries are related to memory management and a further 19.4% are side-channel attacks, suggesting that developers should focus their efforts on systems-level implementation issues. In-depth subclassification reveals that just 14 of 552, or 2.5%, of all vulnerabilities are due to broken protocols or ciphers.

Notably, we find that some components of a cryptographic codebase are disproportionately responsible for producing vulnerabilities: of the vulnerabilities in our dataset for which we were able to obtain location data, over 35% were located in implementations of the SSL/TLS protocols, with a further 27.9% arising from certificate parsing implementations. The median exploitable lifetime of a vulnerability in a cryptographic library is 3.88 years, providing malicious actors a substantial window of exploitation.

Through an analysis of OpenSSL's major version releases, we find that OpenSSL has an average defect density of 1 CVE per thousand lines of code. In our case study of OpenSSL forks, we provide empirical evidence suggesting that debloating a cryptographic codebase materially improves its security. Taken together, these findings support long-held anecdotal beliefs in the security community that cryptography is particularly tricky to implement in practice and provide actionable steps forward for developers working on these libraries and other cryptographic software.

Overall, our primary contributions are as follows:

- After thorough manual review and cleansing, we compile and publish a dataset of all vulnerabilities in cryptographic libraries to date.
- We perform an in-depth review of each issue, extensively characterizing all vulnerabilities by type, feature location, lifetime, severity, and other characteristics.
- We further investigate complexity within the library codebases as a potential source of vulnerabilities, finding substantial variation in complexity among different cryptographic components. Among OpenSSL and its forks, we quantify rates of vulnerability introduction and the security impact of excess code removal.
- We present two novel classification taxonomies specific to cryptographic software and provide guidelines to improve the NVD's data quality.

The rest of the paper is organized as follows: We begin in §2 with a brief discussion of related work examining cryptographic software, vulnerability life cycles, and the relationship between complexity and security. In §3, we describe our process for selecting systems and data sources in more detail and discuss mitigation strategies for inconsistencies in vulnerability data reporting. We then describe the properties of vulnerabilities in cryptographic libraries, measure the complexity of these libraries, and discuss its impact on security (§4). We conclude in §5 and §6 with a discussion of takeaways for developers and users of cryptographic libraries and implications for future development.

2 RELATED WORK

Relationship Between Complexity and Security: There have been several studies broadly investigating software complexity and its effect on security in general-purpose, non-cryptographic software. Ozment et al. [84] 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 that period. They found that vulnerabilities live in the OpenBSD codebase for over two years on average before being discovered and patched. Zimmermann et al. [96] similarly analyzed vulnerability correlation with various software metrics in the Windows Vista operating system, finding a weak correlation. More recently, Azad et al. [62] studied the effects of software debloating in web applications, observing how removing significant percentages of the codebase in PHP web applications impacted vulnerabilities present. In contrast to prior work, we focus specifically on cryptographic software and the vulnerabilities it produces.

Empirical Vulnerability Analysis: While there is a large existing body of work studying vulnerability life cycles and patches, prior work has not included cryptographic software in their datasets. Li et al. [76] and Shahzad et al. [89] both conducted large-scale analyses of vulnerability characteristics in non-cryptographic open-source software, including various operating systems and web browsers. Rescorla et al. [87] 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.

Lazar et al. [75] studied the product sources of 269 cryptographic vulnerabilities in the NVD, 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. In contrast to Lazar et al., we draw an important distinction between cryptographic software and cryptographic vulnerabilities: cryptographic vulnerabilities must necessarily originate in the source code or usage of cryptographic software,[1] but cryptographic software produces a broader class of vulnerabilities that are not only cryptographic in nature (e.g., Heartbleed, a systems-level vulnerability caused by a missing bounds check, was excluded from Lazar et al.'s study). In our work, we seek to fill this gap in prior work and empirically quantify the practical security outcomes of cryptographic implementations in substantially greater depth, including studying of the complexity of the source code and how that further impacts security.

Walden [94] conducted a case study of OpenSSL's software development practices in the aftermath of Heartbleed, analyzing changes made to the codebase and observing that OpenSSL adopted additional recommended best practices, including reducing the size of the codebase, in the wake of the breach. Walden did not study vulnerabilities within OpenSSL and instead focused primarily on general codebase metrics surveying project activity and changes to the codebase. In this work, we conduct the first large-scale, comprehensive analysis of complexity and vulnerability trends across multiple cryptographic libraries.

Role of Human Factors in Software Insecurity: A growing body of work considers why even experienced developers struggle to write secure software, particularly when relying on external APIs [79, 83, 93], and investigates how organizations can better support developers and prevent mistakes [72, 85]. Other ethnographic studies have focused specifically on the usability of cryptographic libraries, finding that the software and documentation of these libraries are opaque and difficult for non-specialists to understand [58, 73, 78, 86]. 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. In this work, we focus on security *within* the API, a question orthogonal to security issues deriving from the *misuse* of the API.

3 METHODOLOGY

To study the causes and characteristics of vulnerabilities in cryptographic software, we collect source code repository and vulnerability data from 23 open-source cryptographic libraries. For the purpose of this work, we define a *cryptographic library* as a general-purpose collection of implementations of cryptographic primitives and/or protocols. In particular, we exclude libraries that act only as bridges to libraries in other languages ("wrappers") and other libraries that exist primarily to offer an interface for another library, as well as libraries focused on comparatively niche primitives such as multi-party computation and zk-SNARKs.

§3.1 describes our process for selecting libraries to include in the study. We discuss the vulnerability characteristics studied and our collection and data cleansing methodology for each in §3.2 - §3.5. While we collected some of our repository and vulnerability data through automated web scraping, due to inconsistencies and inaccuracies in the NVD and other sources the majority of our dataset was manually compiled and analyzed. §3.6 describes our criteria for selecting complexity metrics, and §3.7 summarizes steps take to mitigate the limitations of the work.

Table 1: Cryptographic Libraries—The 23 cryptographic libraries studied, listed in order of total CVEs published in each library from 2005 through 2022.

	Cryptography Library	Primary Language	CVE Count
1.	OpenSSL	C	203
2.	GnuTLS	C	62
3.	Mozilla NSS	C	54
4.	WolfSSL	C	54
5.	Mbed TLS	C	49
6.	Botan	C++	26
7.	Bouncy Castle	Java	22
8.	MatrixSSL	C	21
9.	Libgcrypt	C	17
10.	LibreSSL	C	9
11.	Crypto++	C++	9
12.	Nettle	C	7
13.	PyCrypto	Python	5
14.	Python-Cryptography	Python	3
15.	LibTomCrypt	C	4
19.	Golang Cryptography	Go	3
16.	Relic	C	2
17.	Sodium Oxide	Rust	2
18.	BoringSSL	C	2
20.	Rustls	Rust	1
21.	CryptLib	C	1
22.	Orion	Rust	1
23.	PyCryptodome	Python	1
			Total: 552

In the interest of open access, we have publicly released all data used in this analysis.[2]

3.1 Systems Analyzed

We select cryptographic libraries 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) **Sufficient CVE Reporting:** Since we use the NVD as our vulnerability source, all cryptographic libraries included in the study must have at least one reported CVE. We necessarily exclude a small number of cryptographic libraries, such as libsodium and Ring, because they have no recorded entries in the NVD (as shown in Table 7). Throughout the paper, for certain sub-experiments (such as calculating lifetime) we

[1] The exception to this statement is 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.

[2] https://github.com/jenny-blessing/cryptographic-libraries

further select for systems that have sufficient quantity or quality of reported data to allow us to make generalizable conclusions.

As shown in Appendix A, 23 of the 37 cryptographic libraries we examined satisfy both of these requirements. Table 1 contains these 23 libraries included along with their respective CVE counts.[3]

While we list CVE counts of each individual library to illustrate the composition of our dataset, we stress that absolute vulnerability counts as reported in the NVD are not an effective measure of a system's security and that the listing in Table 1 should *not* be interpreted as a ranking of library security–indeed, a high CVE count can be an indicator of transparency and robust security analysis. We discuss selection bias in the NVD in greater detail in sections 3.7.1 and 4.4.

3.2 Vulnerability Data Sources

We use the NVD [81], managed by the National Institute of Standards and Technology (NIST), as a standardized source of vulnerabilities from which to construct our dataset. For the purposes of this work, we define vulnerability as an entry in the Common Vulnerabilities and Exposures (CVE) list maintained by MITRE [66]. While individual product bug trackers often provide more granularity, they do not allow us to standardize or compare across systems as the NVD does.

When a new CVE is created, the NVD assigns a severity score (CVSS) [82] and performs additional analysis before adding the vulnerability to the database. We scrape CVE data from a combination of the official NVD site and a third-party platform, CVE Details [20], since CVE Details organizes CVEs by product and vendor, enabling us to retrieve all CVEs for a specific system, but is missing certain CVE information.

To supplement and refine the data provided by the NVD, we manually review individual projects' bug trackers, mailing lists, blogs, and other external references (e.g., academic papers presenting the attacks) for more granular information on CVEs (e.g., patch commit description, team discussion of a particular vulnerability, etc.) and feature changes to a library over time. We were able to obtain additional insights and relevant information in this way, particularly from individual project issue trackers (for instance, we collected security ratings as determined by the project rather than the NVD, whether the CVE was patched, and internal analysis of the problem), which we use to construct our dataset. This process naturally varies by project based on the quality of its security advisories and the information available to the public. We also recover an additional ($n = 6$) CVEs erroneously listed only under parent projects in the NVD but discussed in project security advisories (e.g., a CVE in BoringSSL was listed under Chrome, and a small number of Mozilla NSS CVEs were filed under Firefox).

In total, our dataset consists of $n = 552$ CVEs in cryptographic libraries published by the NVD between 2005 and 2022, inclusive.

3.3 Classifying Vulnerability Type

The NVD assigns each vulnerability a Common Weakness Enumeration (CWE) [65] broadly classifying the vulnerability type. In an initial review of CWE labels, however, we observe multiple issues in the context of our study:

1. **Missing CWEs:** 60 CVEs, or just over 10% of our dataset, had no CWE label.
2. **Overly broad categorizations:** We found that the CWE labels assigned were often overly broad and vague regarding the cause of the issue (e.g., "Improper Input Validation," which was the CWE label for 29 CVEs in our dataset and which represented everything from a buffer overread to weak parameter values for a particular cipher to a command injection attack from lack of shell input sanitization in a configuration script. Indeed, the 2006 Bleichenbacher signature forgery attack was included under this category.
3. **Inconsistent labeling:** Similar issues can be assigned semantically different categories depending on the individual who categorized them, particularly for comparatively ambiguous issues comprised of a chain of errors (e.g., an integer overflow causing a buffer overflow, which we observed categorized as a "Numeric Issue" in one CVE and a "Buffer Overflow" in another). This is a particular concern for side-channel attacks, where we count 14 different CWEs used to describe this category of attacks within our dataset.

We describe issues encountered with the quality of the NVD and suggested improvements in greater detail in §4.4.

3.3.1 Classification Taxonomy. We find sufficient inconsistencies and discrepancies in the official NVD labeling that we develop a new classification taxonomy customized to issues found in cryptographic software and manually reclassify all CVEs. Our taxonomy consists of six broad categories described in Table §8 in the Appendix. This manual labeling has several benefits, allowing us to include CVEs that did not have labels in the NVD and, most importantly, to provide a far more precise categorization better suited to cryptographic software, including sub-classifications within each category.

We describe our specific methodology below:

1. We begin by reviewing the official NVD description and manually searching for an individual project security advisory and/or bug discussion thread as available. The NVD's external references section is often useful here, though we found it hampered by broken links (particularly for older CVEs). In these cases and in cases where no patch commit was included, we manually search for the CVE ID or other relevant keywords in the source repository.
2. For $n = 359$ CVEs where we were able to obtain the patch commit (roughly two-thirds of the dataset), we review the changes in depth. If we did not have access to the patch commit, we generally defaulted to the categorization equivalent to the existing CWE label unless the NVD and/or project description strongly suggested otherwise.
3. Once we have collected and reviewed all relevant information, we map the CVE into the existing taxonomy according to its root cause (e.g., what went wrong in the design or

[3] Note that because four CVEs in the dataset are filed under more than one library in the NVD, the total CVE count of 552 is slightly less than the sum of the individual library counts since we exclude duplicate entries.

implementation). There is some degree of ambiguity here: vulnerabilities are sometimes composed of a chain of errors, causing an inevitable degree of overlap among categories. In these cases, we select the category that best describes the *initial* problem that started the chain (e.g., an integer overflow that led to a buffer overflow would be considered a Numeric Issue). If there is a lack of adequate information available or in particularly ambiguous cases, we again err on the side of defaulting to the NVD classification.

(4) A second, independent labeler with domain knowledge reviewed $n = 71$ cases of conflict where our taxonomy differed semantically from the official NVD label, without being given the first labeler's decision or the original NVD label. In the case of disagreement between the labelers, the labelers discussed until consensus was reached and revised the taxonomy accordingly as needed.

Memory Safety Flag: During our in-depth review we find a number of vulnerabilities whose root cause may not have been a memory-related issue, but which were exacerbated or made exploitable by the use of a memory-unsafe language—for instance, the example mentioned above where an integer overflow led to a buffer overflow. To account for this nuance we add an additional binary flag to every CVE in a C/C++ codebase indicating whether the problem would have been mitigated and/or eliminated by memory safety. The combination of the root cause classification and the memory safety indicator allow us to better characterize these vulnerabilities.

3.3.2 Calculating Vulnerability Lifetime. We define a vulnerability's lifetime as its *exploitable* lifetime, i.e., the period of time from when the source code in which the flaw exists is released to the patch for the vulnerability is released. Calculating a lower bound on this lifetime requires determining the release date of the first version affected and the release date of the patch, consistent with methodology used in prior work [84].

We scrape affected versions from the NVD as well as collecting versions from project security advisories as available. We observe that NVD descriptions in practice generally only include the patch version, describing the affected versions as "version X and before." In these cases, if we cannot verify that this is a foundational vulnerability from the project's own security advisory, we mark the version introduced as unknown and exclude it from lifetime calculations. After thorough manual review, we identify four systems (OpenSSL, GnuTLS, Mozilla NSS, and Botan) that consistently report both the initial and patch versions in project security advisories or bug trackers.

Mapping Versions to Release Dates: Once we know the initial and patch versions for a CVE, we determine the release dates of those versions in order to calculate lifetime. For each system we study, we construct a dataset of versions and release dates by manually reviewing 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, and required substantial manual trawling of various mailing lists. In three cases, we were only able to find the month and year of a vulnerability's release date. In these cases, we used the 15th of the month as an approximation of the release date.

3.4 Calculating Vulnerability Severity

We use the NVD CVSS scores to study vulnerability severity across systems. There are a few considerations here since the NVD revised the CVSS classification in 2015 halfway through the timeframe of our study, creating CVSS v3 (as opposed to the previous CVSS v2). Since the NVD's policy is not to retroactively score vulnerabilities published prior to December 20, 2015 [21], the majority of vulnerabilities in our dataset only have v2 scoring, while more recent vulnerabilities only have v3 scoring. To maintain consistency, we use v2 scoring where available in our calculations, and use v3 only if no v2 score is provided.

3.5 Determining Vulnerability Location

To track the location of vulnerabilities within the codebase, we collect the file path(s) within each patch commit. We were able to recover patches for $n = 359$ CVEs, and so we have location information for two-thirds of the dataset. If we were unable to obtain a patch commit or file path for a particular vulnerability, we exclude it from location experiments. Since the file path collected is from the library version at the time and several projects have been refactored over the years in, we label each file path according to the general location category, as shown in our dataset.

3.6 Complexity Metrics

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. Prior work [90, 96] has shown that these two metrics are among the best complexity predictors of vulnerabilities in non-cryptographic software. We define cyclomatic complexity as the number of linearly independent paths through a system's source code, following McCabe's 1976 definition [77].

Lines of Code: We use the command-line tool `cloc` [67] to count the total lines of code for each language in a codebase. Throughout our study, we only count source code lines in the primary implementation language of a library (as referenced in Table 1) and exclude blank lines, comment lines, and header files. We collect LOC measurements of the cryptographic systems over time as well as for specific version releases.

Cyclomatic Complexity: We use a separate command-line tool, `lizard` [95], to calculate the cyclomatic complexity of each individual function. When comparing results across languages, we use a modified version of the Lizard output to better standardize result values. Specifically, we exclude one-line functions (such as getters and setters in object-oriented languages) from our calculations. We also exclude test code and other components that would not be included in the compiled binary. We additionally calculate the average cyclomatic complexity number (CCN) for a given set of files by taking the average over all the functions, rather than calculating the CCN of each file and averaging the files together as Lizard does.

3.7 Limitations

Here, we summarize the limitations of our vulnerability and system datasets and describe steps taken to mitigate these limitations where possible.

*3.7.1 **NVD Vulnerability Reporting**.*

Reporting Bias: The NVD suffers from selection bias in that not all systems report vulnerabilities when discovered. Some vendors pay little attention to the NVD database and do not bother to register vulnerabilities as CVEs, and others skew towards only reporting high-severity CVEs [60].

To mitigate reporting inconsistencies across systems, we avoid using CVE count as an absolute metric in our analysis since this is likely a better indicator of a vendor's reporting practices than of a system's security.

Quality Bias: Even when a system has an appropriate distribution of CVEs reported, the CVE listings and/or project security advisories often fail to include sufficient detail. For instance, as discussed in §3.3.2, most systems do not accurately report versions affected by a CVE. We mitigate this primarily through extensive manual review to supplement and cleanse the dataset.

While manual review is essential to ensuring the consistency and accuracy of the dataset, it can also introduce ambiguity around the classification, an inherent limitation of manual analysis. We control for this through having two independent reviewers review any labeling in conflict with the NVD, and by providing a detailed, public dataset for independent validation of our results.

*3.7.2 **Systems Studied**.*

Open-Source: All systems studied are open-source projects and it is possible that the trends we observe will not be present in proprietary, closed-source software. In order to accurately measure complexity, though, we find it necessary to focus solely on open-source systems, making this limitation unavoidable.

Lack of Language Diversity: Of the 23 cryptographic libraries studied, 15 were written in C or C++, with the remaining 8 written in either Java, Python, Rust, or Go. The high percentage of cryptographic libraries written in C/C++ reflects a historical tendency by developers to default to C++ as the language of choice for high-performance systems software, and this is an inherent limitation of the population size, not the sample size.

Non-Technical Factors: The security of a system is impacted by many economic and human factors in addition to codebase complexity and other software metrics. Software development and testing practices, developer experience level, project funding, and other considerations all affect the quantities and categories of vulnerabilities introduced but are not reflected in codebase data.

4 RESULTS

Here, we present our empirical analysis of the causes and characteristics of vulnerabilities in cryptographic software. We begin by exploring several qualitative and quantitative properties of vulnerabilities discovered in cryptographic libraries. In the second half of our analysis, we explore complexity within cryptographic software, including how the complexity of different features varies. and quantify complexity's impact on security outcomes.

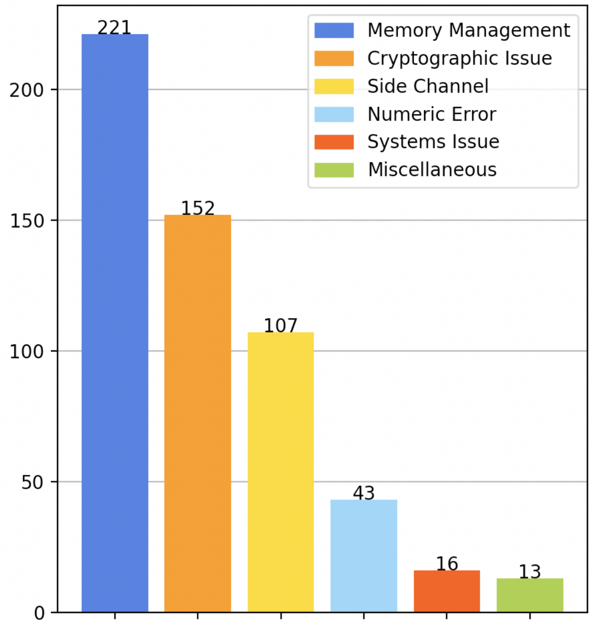

Figure 1: Vulnerability Types — Taxonomy of vulnerabilities by root cause across six main categories within cryptographic software. Table 8 contains descriptions of the taxonomy used and examples of vulnerabilities from each category.

Table 2: Categorizations of Cryptographic Vulnerabilities—Each subcategorization corresponds to a common vulnerability type found among the 152 CVEs within the 'Cryptographic Issue' category.

Cryptographic Subcategories	CVE Count
Certificate Verification Logic Error	41 (26.8%)
TLS/SSL Logic Error	33 (21.7%)
Insufficient Parameter Validation	21 (13.8%)
Insufficient Randomness	13 (8.6%)
Incorrect Cipher Implementation	12 (8.0%)
Protocol Attack	7 (4.6%)
Support for Broken or Risky Algorithm	7 (4.7%)
Miscellaneous	18 (11.8%)

4.1 Characteristics of Vulnerabilities in Cryptographic Software

In this section, we characterize the 552 vulnerabilities affecting the cryptographic libraries studied. We first categorize vulnerabilities

Table 3: Distribution of CVEs among the various components of crypto software — Of the 359 CVEs in our dataset for which we were able to find a patch commit in the library's main language (and therefore identify the location), over one-third originated in SSL/TLS implementations.

Source Code Component	CVE Count
SSL/TLS	129 (35.9%)
X.509 (including ASN.1)	100 (27.9%)
Encryption/Signature Algorithms	65 (18.1%)
Large Integer Implementations	18 (5.0%)
Other Primitives	11 (3.1%)
Other Protocols	8 (2.2%)
Miscellaneous (RNG, UTF encoding, etc.)	28 (7.8%)

by type, including broadly classifying them as cryptographic or non-cryptographic in nature, and investigate origin within the source code. We further study the severity and exploitable lifetime of these vulnerabilities.

4.1.1 Vulnerability Type. Figure 1 shows the respective CVE counts for each of the six main vulnerability types (see §8 for detailed descriptions of each). As discussed in §4.1.1, because of the lack of consistency and specificity in NVD type labeling, we manually recategorize the CVEs in our dataset based on review of CVE descriptions and patch commits according to the taxonomy shown.

General memory management issues comprise the largest individual category at 221 out of 552, or 40.0%, of CVEs. This includes 202 memory safety issues (out-of-bounds write, out-of-bounds read, incorrect calculation of buffer size, etc.) as well as 19 other memory-related issues such as infinite recursion and memory exhaustion.

Cryptographic issues comprise the second-largest individual category at 27.5%. We further subdivide this category §4.1.2 and so will not dwell on it here. Various side-channel attacks, such as timing or memory-cache attacks, further produce 19.4% of CVEs in cryptographic libraries. For the purpose of Figure 1 we list side-channel attacks separately from the general "cryptographic issues" category since they exploit weaknesses in the physical hardware of a system or timing and cache-access data rather than direct flaws in the cryptographic implementations, but side-channel vulnerabilities can also broadly be considered to be cryptographic in nature. We will revisit this distinction in 4.1.2 when we present a more granular categorization of cryptographic vulnerabilities.

A further 7.8% of vulnerabilities arise from numeric errors (i.e., errors in numerical calculation or conversion not specific to any one cipher or algorithm, such as carry propagating errors or squaring very large numbers), and various systems issues comprise 2.9%.

Memory Safety Issues in C/C++ Code. Figure 1 displays root cause vulnerability data across all libraries of all languages, but there is an additional question of interest here: within libraries written in C/C++, how many vulnerabilities are either caused *or* exacerbated by use of a memory-unsafe language? Using the memory safety flag described in §3.3.1, we find that 248 out of 512 C/C++ CVEs, or 48.4% of the CVEs in C/C++ libraries, would have been either prevented or mitigated by using a different language.

Table 4: Exploitable Lifetimes—Exploitable lifetimes (in years) of vulnerabilities in the four cryptographic libraries with version reporting data (see Table 7).

System	# CVEs	Median Lifetime	Avg. Lifetime	StdDev Lifetime
OpenSSL	189	3.78	4.1	3.06
GnuTLS	16	1.70	1.91	1.46
Mozilla NSS	30	11.25	8.42	5.55
Botan	23	4.12	6.25	5.97

4.1.2 *Cryptographic Vulnerabilities.*

Our findings show that just 27.5% of CVEs in cryptographic software are directly related to the cryptographic design and implementation. To verify that this trend is consistent across libraries (and not due to skewed data from any one particular library), we calculate the ratio of cryptographic to non-cryptographic CVEs in the five cryptographic libraries with the largest quantities from Table 1, finding that the individual library percentages are consistent within the range of 25-35%. Since a further 19.5% are side-channel vulnerabilities, the total percentage of vulnerabilities broadly related to the cryptographic nature of the source code is 47%, or almost half.

Table 2 shows the particular subtypes of cryptographic design and implementation issues in greater detail. Most notably, we find that just 9.2% of cryptographic CVEs ("Protocol Attack" and "Support for Broken or Risky Algorithm") are due to a flaw in a theoretical algorithm or protocol specification. The clear majority are caused by various errors in individual library implementation, particularly in the TLS state machine logic and certificate verification. Certificate verification source code (including X.509 ASN.1 parsing) is particularly tricky to get right, with implementation logic errors producing over a quarter of all cryptographic issues. The miscellaneous category further includes a small number of CVEs for which there was insufficient information to make a more detailed assessment.

4.1.3 Vulnerability Source. Which components of a cryptographic library produce high numbers of vulnerabilities? This is a related, but distinct, question from the discussion in §4.1.2 around categorizing cryptographic issues. Here, we are interested in what areas of the codebase frequently produce *all* types of vulnerabilities. We select the 367 CVEs (out of our dataset of 552) for which we can find a patch commit, and thereby identify the location of a CVE within a codebase.

Figure 3 shows the distribution of CVEs among library components. TLS/SSL protocol implementations produce over 1/3 of CVEs, while encryption and digital signature algorithms produce 18.1% of CVEs and various primitives produce just 3.1% of CVEs. Certificate parsing is also noteworthy, with X.509 and ASN.1 implementations collectively producing around 27.9% of CVEs.

4.1.4 Vulnerability Lifetime. Table 4 displays the median and average exploitable lifetimes for four systems along with the sample standard deviation. As previously discussed in §3.3.2, we are only

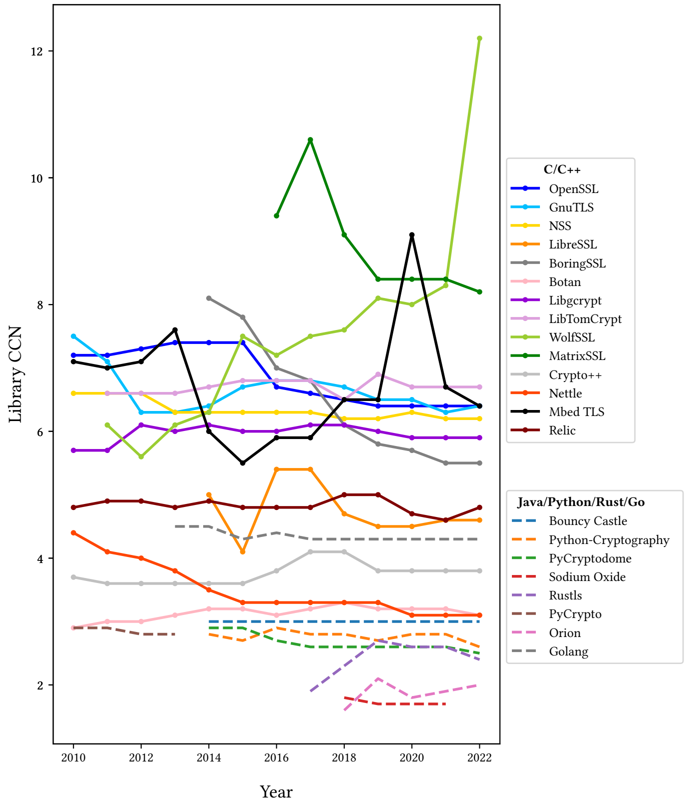

Figure 2: Overall Complexity Over Time—Average CCN of all 23 cryptographic libraries from 2010 through 2022. Libraries written primarily in C/C++ are indicated with a solid line, while all other languages (Java, Python, Rust, and Go) are shown with a dashed line. Several codebases were either not under active development or were otherwise unavailable for part of our range of study, and are therefore only shown for a subset of the time period.

able to obtain accurate version reporting data for four of the cryptographic libraries studied (n = 258 CVEs in total), and so we calculate lifetimes for these systems only.

We find that overall, the median lifetime of a vulnerability in cryptographic software is 3.85 years with a standard deviation of 4.04 years, providing malicious actors a substantial window of exploitation. In fact, because clients frequently continue using outdated versions without updating (even in the wake of a major security breach [69]), these calculations represent a lower bound on the actual exploitable lifetime. Moreover, we necessarily include only vulnerabilities that have been discovered and reported, so vulnerabilities lying undiscovered in the codebases may further increase the average window of exploitation.

4.1.5 Vulnerability Severity. The average severity score of CVEs in our dataset was 5.37, with a standard deviation of 1.80. Qualitatively, we observed very little variation among libraries in vulnerability severity, suggesting that cryptographic libraries are reporting a representative range of vulnerabilities, not merely the most severe.

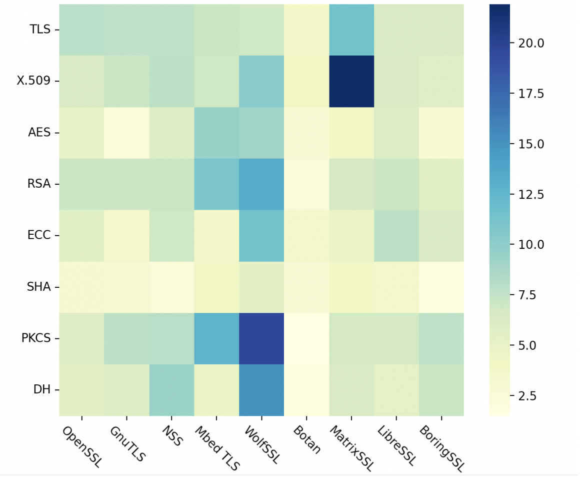

Figure 3: Individual Feature Complexity—Variations in complexity of nine feature common to cryptographic software among libraries implementing these features, with darker colors corresponding to a higher complexity. Measurements were taken from each codebase in mid-2023.

4.2 Code Complexity and Security

Excessive complexity is often cited within the security community as the reason for adverse security outcomes. In this section, we explore this association by studying sources of complexity within cryptographic libraries and how this complexity impacts security.

First, we study the cyclomatic complexity of libraries over time, observing that a project's primary language is strongly correlated with code complexity. We further examine how this complexity breaks down among different features within the library. Second, we study the rate of CVE introduction per LOC in OpenSSL to approximate a lower bound for CVE frequency. Finally, 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.

4.2.1 Overall Library Complexity. Figure 3 shows the cyclomatic complexities of all cryptographic libraries studied over the 12-year period from 2010 through 2022. We observe significant variation in structural code complexity across systems, with the average CCN of the most complex library (MatrixSSL, with an average CCN of 9.0) almost five times as large as the average CCN of the least complex (Sodium Oxide, with an average of 1.74).

The divergence of the results based on language is particularly noteworthy: there is a clear trend of cryptographic libraries written in C or C++ having substantially greater structural complexity compared to codebases written in Java, Python or Rust. In fact, of the 14 libraries studied that were written primarily in C or C++, just two (Botan and Nettle) consistently maintained a complexity level on par with that of non-C libraries.

Table 5: Rate of Vulnerability Introduction—CVEs introduced per thousand lines of C/C++ source code (KLOC) across four distinct versions of OpenSSL.

Major Version	Release Date	Average CCN	Most Recent Minor Version	LOC Change	CVEs Introduced	CVEs / KLOC
1.0.2	1/22/2015	6.3	1.0.1l	22,236	25	1.12
1.0.1	3/14/2012	6.1	1.0.0h	18,766	33	1.76
1.0.0	3/29/2010	6.1	0.9.8n	15,510	12	.77
0.9.8	7/5/2005	6.1	0.9.7g	23,174	28	1.2

4.2.2 Individual Feature Complexity. The high levels of complexity observed in §4.2.1 raise a natural follow-up question: what are the sources of this complexity within these libraries? We select eight features common to a quorum of libraries that collectively cover the typical components of a cryptographic library (e.g., key exchange, certificate parsing, authenticated encryption, digital signatures, etc.). For the purpose of approximating feature complexity, we do not differentiate between different versions of a standard or primitive and opt to include all versions in our measurements. For instance, some implement both SHA-256 and SHA-512 and others implement only one of the two, while still others include older versions such as SHA-1. Similarly, we collectively measure and include all versions of SSL and TLS that a library provides under the 'TLS' heading. We include ASN.1 parsing and certificate revocation logic under X.509, since some projects store these components within the same file, making them challenging to separate.

Figure 3 shows a heatmap visualization of feature complexity across libraries that implemented at least one version of all the features selected. Among feature set, PKCS stands out as having a consistently high relative complexity. Other PKI components, including ASN.1 and X.509 parsing, also exhibit higher than usual complexities. Conversely, the SHA algorithm family is notable for having a very low structural complexity relative to other components.

4.2.3 Relationship Between LOC and Vulnerability Count. Having studied structural complexity within cryptographic software and its sources, we now ask what security outcomes complexity produces. Here, we examine whether there is a correlation between the lines of code introduced in a version and the number of CVEs introduced. To control for variations in vulnerability reporting practices, rather than considering across different systems we instead contrast CVEs introduced across different versions of the same system, OpenSSL. We consider only OpenSSL here because it is the only cryptographic library with sufficient quantity and quality of CVE entries to allow us to do such a comparison. It is also to date still the single most widely used cryptographic library [13].

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 source code for all versions was obtained from the releases stored in OpenSSL's source code repository [42]. 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 represents a better approximation of the impact of the version on the codebase.

Column 5 in Table 5 gives the LOC change for each version, 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 [43]. We estimate the number of CVEs introduced per thousand lines of code by taking the ratio of columns 5 and 6 in Table 5.

Column 7 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 this necessarily includes only vulnerabilities that have been discovered. The ratio of vulnerabilities existing in the codebase per thousand LOC is very likely higher when taking into account vulnerabilities that have not yet been discovered, 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.

4.3 Case Study: OpenSSL, LibreSSL, & BoringSSL

The OpenSSL project and its forks provide a natural experiment allowing us to study the security impact of major changes to a codebase in the wild. The Heartbleed vulnerability [1] gained international attention in April 2014, bringing OpenSSL into the spotlight with it. The increased scrutiny of the OpenSSL codebase in the wake of Heartbleed prompted the creation of two major forks of the codebase: LibreSSL, developed by the OpenBSD project and released on July 11, 2014 [30], and BoringSSL, developed by Google and released on June 20, 2014 [74].

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

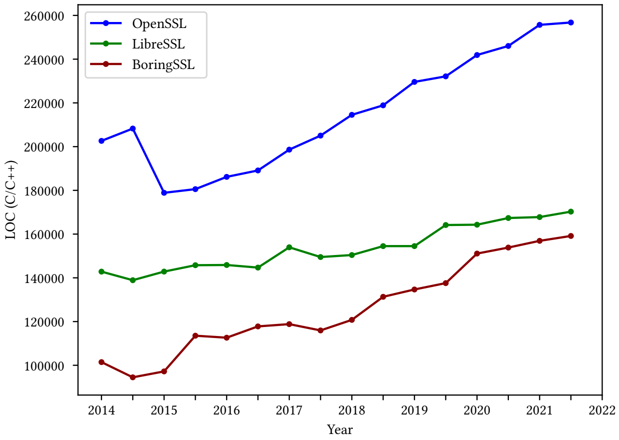

Figure 4: Codebase Growth in OpenSSL, LibreSSL, and BoringSSL. Relative sizes of OpenSSL, LibreSSL, and BoringSSL measured every six months from July 2014 through January 2022. LOC count includes only C and C++ source code (excluding blank lines, comment lines, test code, and header files).

compatibility and portability [26], while BoringSSL was developed for internal use only. OpenBSD forked OpenSSL with the goal of creating a more modern and secure TLS library after a particular disagreement over the way OpenSSL handled memory management [26, 31]. The BoringSSL project, on the other hand, specifically states that the library is not recommended for external projects' use [7]. This difference in stated purpose helps to explain why BoringSSL diverges more from OpenSSL than LibreSSL in overall size (as shown in Figure 4) and features offered.

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 202,227 lines of C and C++ source code in its core library. In the months that followed from early April through June 2014, LibreSSL removed roughly 60,000 C/C++ LOC while BoringSSL removed 100,000 LOC.

Figure 4 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 two forks is primarily due to OpenSSL's maintenance of legacy ciphers and protocols in order to maintain backwards compatibility and portability. 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.

LibreSSL Changes. The OpenBSD team built LibreSSL under the design that the library would only be used on a POSIX-compliant OS with a standard C compiler [31, 92]. This assumption enabled them to remove much of OpenSSL's OS- 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 Kerberos.

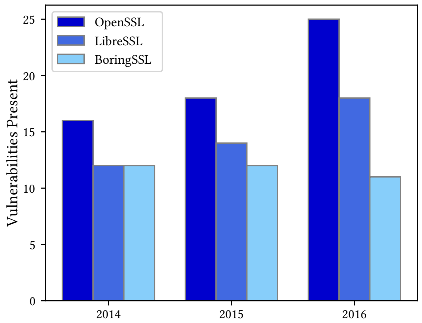

Figure 5: Vulnerability Comparison Post-Heartbleed—Vulnerabilities discovered in OpenSSL after the initial releases of LibreSSL and BoringSSL with a comparison of how many of those vulnerabilities also affected LibreSSL and BoringSSL.

Table 6: Percentages of source code and corresponding CVEs removed in LibreSSL and BoringSSL compared to OpenSSL.

Library	% of Codebase Removed	% of Vulnerabilities Removed
LibreSSL	30%	25% (15/59)
BoringSSL	50%	40.6% (24/59)

BoringSSL Changes. BoringSSL has more limited intended use cases than LibreSSL and so was able to discard roughly twice as much source code 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, Camellia, RC5, MD2, and Kerberos [74]. We also observed that BoringSSL refactored what source code remained more extensively than LibreSSL.

4.3.2 Determining Vulnerabilities Removed. The abrupt jettisoning of approximately 30% and 50% 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? To answer this question, we study vulnerabilities reported in OpenSSL in the wake of Heartbleed and whether they also affected LibreSSL and BoringSSL.

Since we find the official NVD count wholly inaccurate for tracking whether LibreSSL and BoringSSL were affected by CVEs (LibreSSL in fact has a policy of not requesting CVEs[4], as described further in §4.4), we create our own database based on OpenSSL's vulnerability list [43]. For each CVE affecting OpenSSL post-fork, we manually classify it as having also affected LibreSSL or BoringSSL as of July 11, 2014. To do so, we conducted an extensive manual review of commit descriptions in the source code of both libraries [9, 28], the LibreSSL team mailing list archives [29], the LibreSSL change log [27], LibreSSL security advisories [3], and the BoringSSL bug tracker [8].

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. BoringSSL was first released in late June, and so by July 11 both libraries had been released. Vulnerabilities discovered and patched between April 7 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 the offending OpenSSL source code was not present in the library as of 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 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 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 6 shows that of those 59 CVEs, 44 still affected LibreSSL and 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 5 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 show directly that these vulnerabilities were removed from the respective codebases because the original source code was removed, a stronger conclusion than merely demonstrating a correlation between size and security. Furthermore, we empirically quantify the security gains of LibreSSL and BoringSSL as shown in Table 6, and observe that vulnerabilities removed is closely correlated with source code removed across LibreSSL and BoringSSL.

4.4 CVE Reporting Practices

Since our analysis relies heavily on the quality of the vulnerability reporting in the NVD and we spent substantial time manually reviewing the database, we briefly discuss observed reporting practices in cryptographic systems.

How reliable is the CVE data on cryptographic libraries? *CVE Counts:* From Table 1 we observe that OpenSSL has a far greater number of CVEs than any other cryptographic library, with 203 CVEs published during our timeframe compared to the second-highest count of 62 CVEs in GnuTLS. An important 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.

In our manual review of the NVD, we qualitatively that the OpenSSL project is often used as catch-all repository of vulnerabilities relevant to cryptographic source code. CVEs for well-known protocol attacks, such as Logjam, would often only be filed under two or three of the most widely-used libraries, and sometimes only under OpenSSL. For instance, of the 23 libraries studied, the 2015 “FREAK” protocol exploit was only filed under OpenSSL [2].

Our case study of the evolutions of OpenSSL forks provides a natural test to study how the official NVD vulnerability counts compare. Whereas OpenSSL's absolute vulnerability count was 190 CVEs, LibreSSL and BoringSSL recorded just nine and one CVE(s), 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 §4.3), we confirm that the NVD count of vulnerabilities affecting LibreSSL and BoringSSL is substantially lower than the actual number of vulnerabilities that affected those libraries. This appears due in no small part to formal policies of these libraries not to request CVEs[4]. Overall, this finding reinforces the unreliability of absolute vulnerability counts as an indicator of project security.

CVE Types: As previously mentioned in §4.1.1, while we initially relied on CWE data to classify the remaining vulnerabilities, upon closer inspection we found that CWE labeling was sufficiently inconsistent that we needed to manually review descriptions and commits in order to gain a more accurate picture of what vulnerability classes exist.

Moreover, CWE labels do not map well to the types of issues that arise in cryptographic software. For example, while the NVD provides a label for timing side-channel attacks (“CWE-385: Covert Timing Channel”), this CWE is used for only 5 CVEs, with developers often opting to select a more generic label for timing attacks instead. Moreover, the NVD provides no CWE label for side-channel attacks that exploit other environmental factors. Side-channel attacks are an important attack class unique to cryptographic software, but the lack of standard labeling conventions has result in these vulnerabilities being reported under a variety of generic CWE labels. In fact, we observed 14 unique CWE labels used to categorize side-channel attacks in our dataset.

5 DISCUSSION

We set out to better understand the root causes of insecurity in cryptographic software, including the relationship between software complexity and security and other underlying causes of vulnerabilities in these libraries. Here, we highlight the most noteworthy results and their implications for software development practices.

Need for a systems approach to cryptographic software: Our findings lay bare the discrepancy between the critical role cryptographic libraries hold in securing network traffic and the security posture of these libraries. In Figure 3 we demonstrated the unusually high levels of structural complexity in many of today's cryptographic libraries. This kind of complexity makes these codebases more difficult to test and maintain, which in turn has significant implications for library security. In our analysis, we even find at least 11 instances where the patch for a vulnerability inadvertently introduced a new vulnerability (and this count only includes CVEs where this was noted in a project security advisory or discussion thread).

One of the more interesting trends from Figure 4 is that all three of OpenSSL, LibreSSL, and BoringSSL have been gradually increasing in size since the 2014 fork. Even projects that set out to be minimalist and security-focused, as both LibreSSL and BoringSSL did, naturally accumulate additional source code and features over time. While some projects may need to maintain such complexity for backwards compatibility reasons, our findings suggest that where possible developers need to take special care to guard against such bloat through, for instance, conducting an annual audit of the codebase to remove support for outdated or deprecated ciphers as newer ones are added.

A significant, actionable contribution of this work is to provide benchmarking and comparison data for developers of cryptographic software. While a certain base amount of complexity is unavoidable in cryptographic implementations, we can only begin to understand where that baseline may be through large-scale data collection and comparison across the cryptographic software ecosystem.

Side-channel attacks as a threat vector: Side-channel attacks have often been overlooked by library developers, in part because they frequently require physical hardware access. CryptLib, for instance, disputed a memory-cache side-channel CVE because it "does not include side-channel attacks within its threat model" [80]. Our finding that approximately 20% of vulnerabilities in cryptographic software are side-channel attacks provides empirical backing for developers to devote a level of attention to these types of attacks proportionate to the number of security threats produced.

Causes of vulnerabilities within libraries: Not all feature components produce vulnerabilities at the same rate, as demonstrated by the fact that over 1/3 of vulnerabilities are produced by SSL/TLS source code. The findings in this study provide empirical data for developers to focus their debugging, testing, and fuzzing efforts accordingly. Furthermore, while public discussion of these libraries has often focused on widely-publicized breaks of the TLS/SSL protocols, such attacks make up just 1.3% of vulnerabilities in these libraries. In practice, a library is far more likely to be made insecure by a missing bounds check than by use of a weak protocol.

Shift to memory-safe languages: The majority of cryptographic libraries (and particularly the most widely used libraries) are written in C and/or C++, which have historically been the languages of choice for high-performance software in large part because they allow developers direct control over memory allocation. Our results, however, empirically show that just over half of vulnerabilities are in C/C++ libraries are either caused or made more dangerous by the language's memory allowances, many of which are quite primitive. For example, an off-by-one error when calculating a value later used to determine how much memory to allocate is at its core a simple "numeric error," but such mistakes can have a disastrous impact in a non-memory-safe language.

The abundance of memory safety issues present in cryptographic libraries raises valid questions of whether we should continue to rely on non-memory-safe languages to write security-critical systems. Prioritizing memory safety in the software development process is a core component of the U.S. National Cybersecurity Strategy [55] and the Cybersecurity and Infrastructure Security Agency (CISA)'s ongoing initiative to build systems that are "secure by design" [56]. Several new cryptographic libraries have been created in the past few years, mostly in Rust, a popular memory-safe language, and marketed as alternatives to older C libraries such as OpenSSL and GnuTLS. These libraries are still quite new compared to the older C stalwarts, however, and a substantial amount of auditing and performance testing is still needed before they could be considered for adoption by a major OS or web browser, for instance.

It is important to note that "memory-safe" languages are not a panacea for memory-related vulnerabilities, and, contrary to what their name would imply, produce memory leaks and other issues of their own. Moreover, there are still numerous classes of systems vulnerabilities that cryptographic implementations need to contend with, such as downgrade attacks and various side-channel issues, and that will not be solved merely by using a different language. A major contribution of this work is to quantify the prevalence of side-channel attacks in cryptographic software, a class of attacks Rust will not prevent. Nonetheless, several of the newer cryptographic libraries written in memory-safe languages are promising and should be taken seriously as alternatives to their C counterparts.

Complexity variation across languages: In addition to the potential of memory-safe languages to reduce raw quantities of memory safety bugs, one of our most interesting results is that libraries in our study written in memory-safe languages have a significantly lower cyclomatic complexity than the clear majority of those written in C/C++ (as shown in 3), suggesting that the language itself may be a source of complexity. That a small number of C libraries were able to achieve a very low complexity shows this trend is at least partially due to weak software development practices across some of the C systems, though it is also possible that the need for various memory-related conditionals makes C source code inherently more complex than Python and Rust.

6 CONCLUSION

In this paper, we analyzed the real-world security issues found in modern cryptographic software and its root causes, including (1) characteristics of vulnerabilities in cryptographic software and (2) correlations between various complexity metrics and corresponding vulnerability counts. Overall, our findings support the common intuition that it is dangerous to maintain excess amounts of legacy source code, particular C or C++ code, within cryptographic software, and provide empirical evidence advocating for a greater focus on debloating these libraries. Our results further suggest that the rise of memory-safe languages such as Rust and Go will materially improve security, and developers should consider adopting these languages where possible.

7 ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers who took the time to contribute thoughtful feedback to this paper. Jenny Blessing was supported by the MIT Internet Policy Research Initiative; Michael Specter was supported by Google's Android Security and Privacy REsearch (ASPIRE) fellowship; and Daniel Weitzner was supported, in part, by NSF grant Collaborative Research: DASS: Legally Accountable Cryptographic Computing Systems (LAChS) Award Number: 21315415.

REFERENCES

[1] 2014. *The Heartbleed Bug*. https://heartbleed.com/

[2] 2015. *CVE-2015-0204*. https://nvd.nist.gov/vuln/detail/CVE-2015-0204

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

[4] 2017. Some nginx TLS tests started failing with LibreSSL 2.5.3 (but not with 2.4.4). (2017).

[5] 2020. *CVE-2020-13777*. https://nvd.nist.gov/vuln/detail/CVE-2020-13777/

[6] 2023. *BearSSL*. https://bearssl.org/

[7] 2023. *BoringSSL*. https://boringssl.googlesource.com/boringssl/

[8] 2023. *BoringSSL Bug Tracker*. https://bugs.chromium.org/p/boringssl/issues/list

[9] 2023. *BoringSSL GitHub*. https://github.com/google/boringssl

[10] 2023. *Botan*. https://botan.randombit.net/

[11] Legion of the Bouncy Castle 2023. *The Bouncy Castle Crypto Package For Java*. Legion of the Bouncy Castle. https://github.com/bcgit/bc-java

[12] 2023. *BSAFE*. https://www.dell.com/support/kbdoc/en-uk/000181945/dell-bsafe-product-version-life-cycle

[13] 2023. *Censys*. https://censys.io/

[14] Cloudflare 2023. *CIRCL (Cloudflare Interoperable, Resuable Cryptographic Library)*. Cloudflare. https://github.com/cloudflare/circl

[15] 2023. *The Cryptix Project*. http://www.cryptix.org/

[16] 2023. *CryptLib*. https://www.cryptlib.com/

[17] 2023. *Crypto++*. https://www.cryptopp.com/

[18] 2023. *Crypto-JS*. https://github.com/brix/crypto-js

[19] 2023. *cryptography*. https://pypi.org/project/cryptography/

[20] 2023. *CVE Details*. https://www.cvedetails.com/

[21] National Vulnerability Database 2023. *CVE FAQs*. National Vulnerability Database. https://nvd.nist.gov/general/FAQ-Sections/CVE-FAQs#faqLink10

[22] 2023. *GnuTLS*. https://www.gnutls.org/

[23] 2023. *Golang Cryptography*. https://pkg.go.dev/crypto

[24] 2023. *Java Cryptography Architecture (JCA) Reference Guide*. https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[25] 2023. *Libgcrypt*. https://gnupg.org/software/libgcrypt/index.html

[26] 2023. *LibreSSL*. https://www.libressl.org/

[27] 2023. *LibreSSL ChangeLog*. https://github.com/libressl-portable/portable/blob/master/ChangeLog

[28] 2023. *LibreSSL GitHub*. https://github.com/libressl-portable/openbsd

[29] 2023. *LibreSSL Mailing list ARChives*. https://marc.info/?l=libressl&r=1&w=2

[30] 2023. *LibreSSL Releases*. https://www.libressl.org/releases.html

[31] 2023. *LibreSSl with Bob Beck*. https://www.youtube.com/watch?v=GnBbhXBDmwU

[32] 2023. *Libsodium*. https://doc.libsodium.org/

[33] 2023. *LibTomCrypt*. https://github.com/libtom/libtomcrypt

[34] 2023. *MatrixSSL*. https://github.com/matrixssl/matrixssl

[35] 2023. *Mbed TLS*. https://github.com/Mbed-TLS/mbedtls

[36] 2023. *Microsoft Schannel*. https://docs.microsoft.com/en-us/windows-server/security/tls/tls-ssl-schannel-ssp-overview

[37] 2023. *Monocypher*. https://monocypher.org/

[38] 2023. *Mozilla Network Security Services*. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

[39] 2023. *NaCl*. https://nacl.cr.yp.to/

[40] 2023. *Nettle*. https://github.com/gnutls/nettle

[41] 2023. *OpenSSL*. https://www.openssl.org/

[42] 2023. *OpenSSL Releases*. https://github.com/openssl/openssl/releases

[43] 2023. *OpenSSL Vulnerabilities*. https://www.openssl.org/news/vulnerabilities.html

[44] 2023. *Orion*. https://github.com/orion-rs/orion

[45] 2023. *PyCrypto*. https://github.com/pycrypto/pycrypto

[46] 2023. *PyCryptodome*. https://pypi.org/project/pycryptodome/

[47] 2023. *Rambus TLS Toolkit 4.0*. https://www.rambus.com/security/software-protocols/secure-communication-toolkits/tls-toolkit/

[48] 2023. *Ring*. https://github.com/briansmith/ring

[49] 2023. *RustCrypto*. https://github.com/RustCrypto

[50] 2023. *Rustls*. https://docs.rs/rustls/latest/rustls/

[51] 2023. *s2n*. https://github.com/aws/s2n-tls

[52] 2023. *Sodium Oxide*. https://docs.rs/sodiumoxide/latest/sodiumoxide/

[53] 2023. *Stanford JavaScript Cryptography Library (SJCL)*. https://github.com/bitwiseshiftleft/sjcl/

[54] 2023. *WolfSSL*. https://www.wolfssl.com/

[55] The White House February 2024. *A Path Toward Secure and Measurable Software*. The White House. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[56] CISA October 2023. *Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design*. CISA. https://www.cisa.gov/sites/default/files/2023-10/SecureByDesign_1025_508c.pdf

[57] 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. 2015. Keys Under Doormats: Mandating Insecurity by Requiring Government Access to All Data and Communications. *Journal of Cybersecurity* 1.1 (2015), 69–79.

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

[59] Ross Anderson. 1993. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 215–227.

[60] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2021. Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing* (2021).

[61] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. 2023. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[62] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.

[63] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *Progress in Cryptology–LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings 2*. Springer, 159–176.

[64] Frederik Chang. 2013. Is Your Data on the Healthcare. gov Website Secure?, Written Testimony before the Committee on Science. *Space and Technology, US House of Representatives, November* (2013).

[65] MITRE Corporation. [n. d.]. . CWE: Common Weakness Enumeration.

[66] MITRE Corporation. 2023. *Common Vulnerabilities and Exposures*. https://cve.mitre.org/

[67] Al Danial. 2023. *cloc: Count Lines of Code*. https://github.com/AlDanial/cloc

[68] Amira Dhalla, Yael Grauer, Alex Gaynor, and Josh Aas. 2023. Fireside Chat: The State of Memory Safety. (2023).

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

[70] Alex Gaynor. 2021. Quantifying memory unsafety and reactions to it. (2021).

[71] Peter Gutmann. 2002. Lessons learned in implementing and deploying crypto software. In *11th USENIX Security Symposium (USENIX Security 02)*.

[72] Julie M Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. 2018. " We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*. 357–373.

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

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

[75] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. 1–7.

[76] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.

[77] Thomas J McCabe. 1976. A Complexity Measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[78] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle With Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering*. 935–946.

[79] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why do developers get password storage wrong? A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 311–328.

[80] NVD. [n. d.]. . CVE-2018-12433.

[81] U.S. National Institute of Standards and Technology. 2020. . National Vulnerability Database. https://nvd.nist.gov/home.cfm/

[82] U.S. National Institute of Standards and Technology. 2023. *CVSS Information.* https://nvd.nist.gov/cvss.cfm/

[83] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A DeLong, Justin Cappos, and Yuriy Brun. 2018. {API} Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018).* 315–328.

[84] Andy Ozment and Stuart E Schechter. 2006. Milk or Wine: Does Software Security Improve with Age?. In *USENIX Security Symposium*, Vol. 6.

[85] Hernan Palombo, Armin Ziaie Tabari, Daniel Lende, Jay Ligatti, and Xinming Ou. 2020. An ethnographic understanding of software (in) security and a co-creation model to improve secure software development. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020).* 205–220.

[86] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability Smells: An Analysis of {Developers'} Struggle With Crypto Libraries. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019).* 245–257.

[87] Eric Rescorla. 2005. Is Finding Security Holes a Good Idea? *IEEE Security & Privacy* 3, 1 (2005), 14–19.

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

[89] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. 2012. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 771–781.

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

[91] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62.

[92] Ted Unangst. [n. d.]. *LibreSSL: More Than 30 Days Later.* https://www.openbsd.org/papers/eurobsdcon2014-libressl.html

[93] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20).* 109–126.

[94] James Walden. 2020. 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.* 409–419.

[95] Terry Yin. 2023. . Lizard. https://github.com/terryyin/lizard

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

A APPENDIX

Table 7: Cryptographic Libraries. We present a comprehensive listing of commonly used cryptographic libraries and certain repository characteristics. A ● means a library satisfies the property, a ◐ means it partially satisfies the property, a ○ means it does not, and '-' indicates that it is not applicable or unable to be determined. For each library, we approximate years in active development beginning from the initial public release through the most recent repository update, a process that is ultimately best-effort. We consider a library to be currently active if the project repository and/or website have been updated within the last 12 months.

Library	Primary Implementation Language(s)	Years Active	Open Source	CVE(s) Published Between 2005 - 2022	Reports CVE Affected Versions
BearSSL [6]	C	2016 - 2018	●	○	-
BoringSSL [7]	C, C++	2014 - Present	●	●	○
Botan [10]	C++	2000 - Present	●	●	●
Bouncy Castle [11]	Java, C#	2000 - Present	●	●	○
BSAFE [12]	C, Java, Assembly	1996 - Present	○	●	○
CIRCL [14]	Go	2019 - Present	●	○	-
Cryptix [15]	Java	1995 - 2005	●	○	-
Cryptlib [16]	C	1995 - Present	●	●	○
Crypto++ [17] (also known as libcrypto++)	C++	1995 - Present	●	●	○
Crypto-JS [18]	JavaScript	2013 - Present	●	○	-
GnuTLS [22]	C	2000 - Present	●	●	◐
Golang Cryptography [23] [22]	Go	2012 - Present	●	●	-
Java Cryptography Architecture (JCA) [24]	Java	1997 - Present	●	○	-
Libgcrypt [25]	C	1998 - Present	●	●	○
LibreSSL [26]	C	2014 - Present	●	●	○
Libsodium [32]	C	2013 - Present	●	○	-
LibTomCrypt [33]	C	2001 - Present	●	●	○
MatrixSSL [34] (now the Rambus TLS Toolkit) [47]	C	2004 - 2020	◐[4]	●	○
Mbed TLS [35] (previously PolarSSL)	C	2009 - Present	●	●	○
Monocypher [37]	Rust	2017 - Present	●	○	-
Mozilla Network Security Services (NSS) [38]	C, C++	2000 - Present	●	●	◐
NaCl [39, 63]	C	2008 - 2016	●	○	-
Nettle [40]	C	2001 - Present	●	●	○
Orion [44]	Rust	2018 - Present	●	●	○
OpenSSL [41]	C	1998 - Present	●	●	●
PyCrypto [45]	Python, C	2002 - 2014	●	●	○
PyCryptodome [46]	Python, C	2014 - Present	●	●	○
Python-Cryptography [19]	Python	2014 - Present	●	●	○
Relic [61]	C	2010 - Present	●	●	○

[4] MatrixSSL was acquired by a private company in 2020 and has since become closed source, but since the codebase was open source for many years we include it in our dataset.

Library	Primary Implementation Language(s)	Years Active	Open Source	CVE(s) Published Between 2005 - 2022	Reports CVE Affected Versions
Ring [48]	Rust, Assembly	2016 - Present	●	○	-
RustCrypto [49]	Rust	2017 - Present	●	○	-
Rustls [50]	Rust	2016 - Present	●	●	○
s2n [51]	C	2015 - Present	●	○	-
Schannel [36]	-	2000 - Present	○	●	○
Sodium Oxide [52]	Rust	2018 - 2021	●	●	○
Stanford JavaScript Crypto Library (SJCL) [53]	JavaScript	2009 - 2019	●	○	-
WolfSSL [54] (previously CyaSSL)	C	2006 - Present	●	●	○

Table 8: Type Taxonomy—Taxonomy used to classify vulnerabilities by root cause. In cases of ambiguity, we select the category that best describes the initial cause of the issue (see §3.3).

Category	Description	Examples
Cryptographic Issue	Vulnerabilities arising from a direct flaw in the design or implementation of cryptographic primitives, protocols, and algorithms (i.e., the cryptographic nature of the source code).	Implementation does not follow the recommended RFC practice, using a weak or broken protocol, protocol downgrade attacks, use of unsafe primes to generate cipher parameters, allowing incorrect parameter input for a cipher based on cipher spec (e.g., one parameter should always be larger than the other).
Memory Management	Vulnerabilities caused by memory management—namely, allocating, reading and writing, and freeing data. This category also includes issues that can affect non-memory-safe languages, such as infinite loops and other memory exhaustion issues.	Source code does not check whether an input is null prior to access, missing bounds check while reading a buffer (such as a field from an X.509 certificate).
Side-Channel Attack	An attack that exploits variations in time taken to complete an operation or physical hardware, including timing and memory-cache attacks.	Non-constant scalar multiplication, padding oracle attack.
Numeric Error	Incorrect numeric calculations or conversions, particularly in large number arithmetic, that are not specific to any one cryptographic cipher or algorithm.	Integer overflow, carry propagating bug, bit calculation errors, integer type size variations on different architectures.
Systems Issue	Vulnerabilities arising from library interactions with the local OS.	Files stored in a writable sub-directory, command injection in a shell script, thread safety/concurrency issues.
Miscellaneous	All other issues that don't fit into one of the above categories.	Wildcard string matching (e.g., matching two domain names), missing documentation, and miscellaneous other implementation bugs.