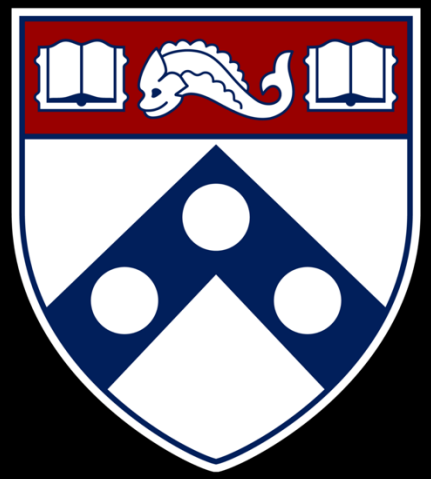
Empirical Security & Privacy, for Humans

UPenn CIS 7000-010

11/20/2025



How Long Do Vulnerabilities Live in the Code?

A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes

Reading

How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes

Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube and Max Mühlhäuser
Telecooperation Lab, Technical University of Darmstadt, Germany

Abstract

How long do vulnerabilities live in the repositories of large, evolving projects? Although the question has been identified as an interesting problem by the software community in online forums, it has not been investigated yet in adequate depth and scale, since the process of identifying the exact point in time when a vulnerability was introduced is particularly cumbersome. In this paper, we provide an automatic approach for accurately estimating how long vulnerabilities remain in the code (their *lifetimes*). Our method relies on the observation that while it is difficult to pinpoint the exact point of introduction for one vulnerability, it is possible to accurately estimate the average lifetime of a large enough sample of vulnerabilities, via a heuristic approach.

With our approach, we perform the first large-scale measurement of Free and Open Source Software vulnerability lifetimes, going beyond approaches estimating lower bounds prevalent in previous research. We find that the average lifetime of a vulnerability is around 4 years, varying significantly between projects (~2 years for Chromium, ~7 years for OpenSSL). The distribution of lifetimes can be approximately described by an exponential distribution. There are no statistically significant differences between the lifetimes of different vulnerability types when considering specific projects. Vulnerabilities are getting older, as the average lifetime of fixed vulnerabilities in a given year increases over time, influenced by the overall increase of code age. However, they live less than non-vulnerable code, with an increasing spread over time for some projects, suggesting a notion of maturity that can be considered an indicator of quality. While the introduction of fuzzers does not significantly reduce the lifetimes of memory-related vulnerabilities, further research is needed to better understand and quantify the impact of fuzzers and other tools on vulnerability lifetimes and on the security of codebases.

1 Introduction

Software flaws that can potentially be exploited by an adversary are referred to as *security bugs* or *vulnerabilities*. Reducing the number of vulnerabilities in software by finding existing ones and avoiding the introduction of new ones (e.g. by employing secure coding practices or formal verification techniques) is one of the primary pursuits of computer security.

Measurement studies on the different stages of the *vulnerability lifecycle* play an important role in this pursuit, as they help us better understand the impact of security efforts and improve software security practices and workflows. The community has produced a number of such outputs in recent years [5, 8, 14, 18, 23, 28, 36]. A vulnerability's lifecycle, or *window of exposure* as introduced by Schneier [34], describes the phases between the introduction of a vulnerability in the code, and the point in time when all systems affected by that vulnerability have been patched. There have been several adaptations of the vulnerability lifecycle concept (e.g. w.r.t. the number of phases or their ordering and its non-linearity), but the general concept remains the same, and a simple version is shown in Fig 1. The lifecycle of a vulnerability (or

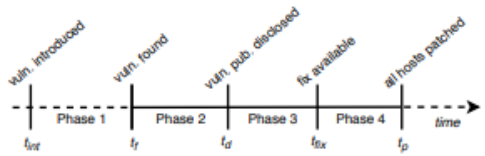
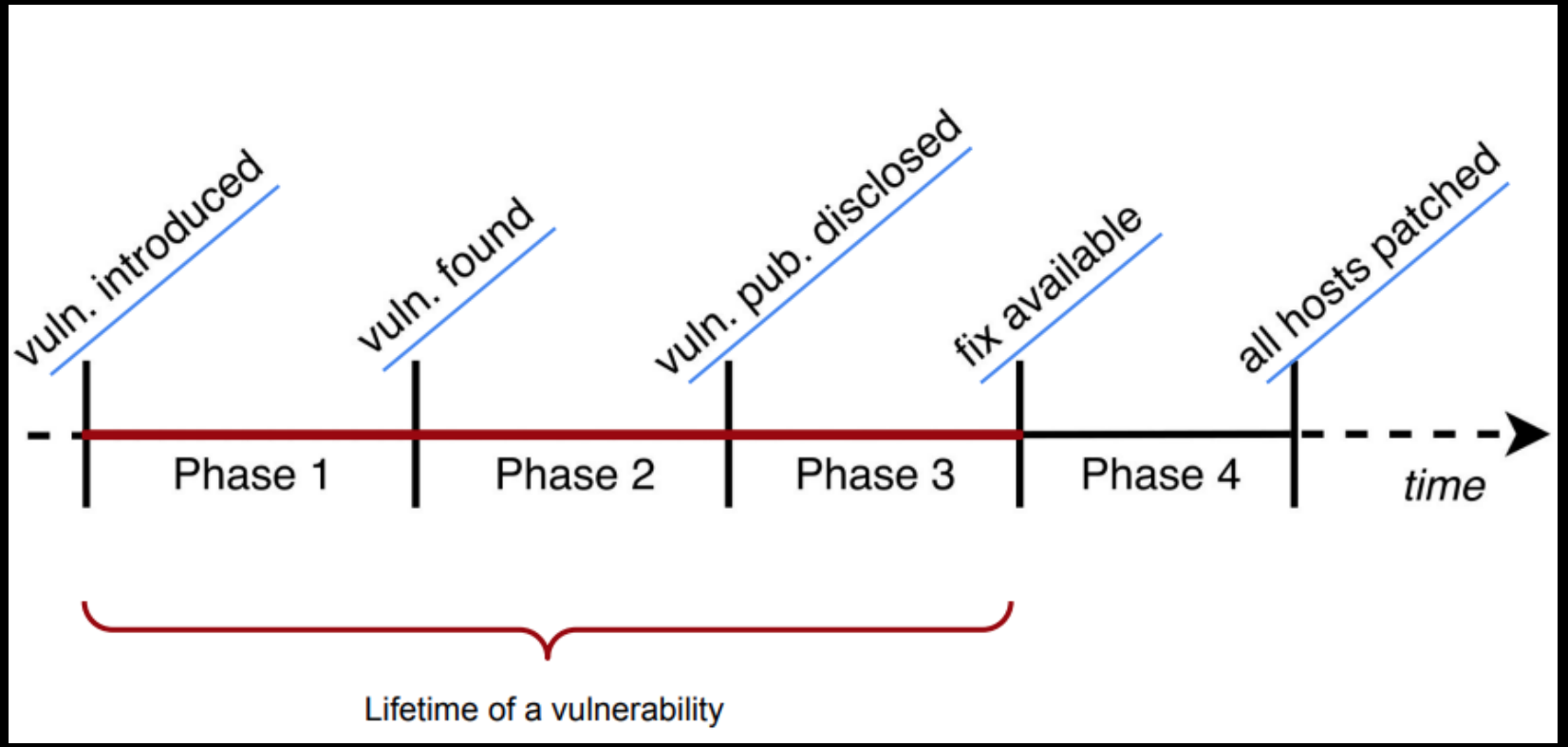


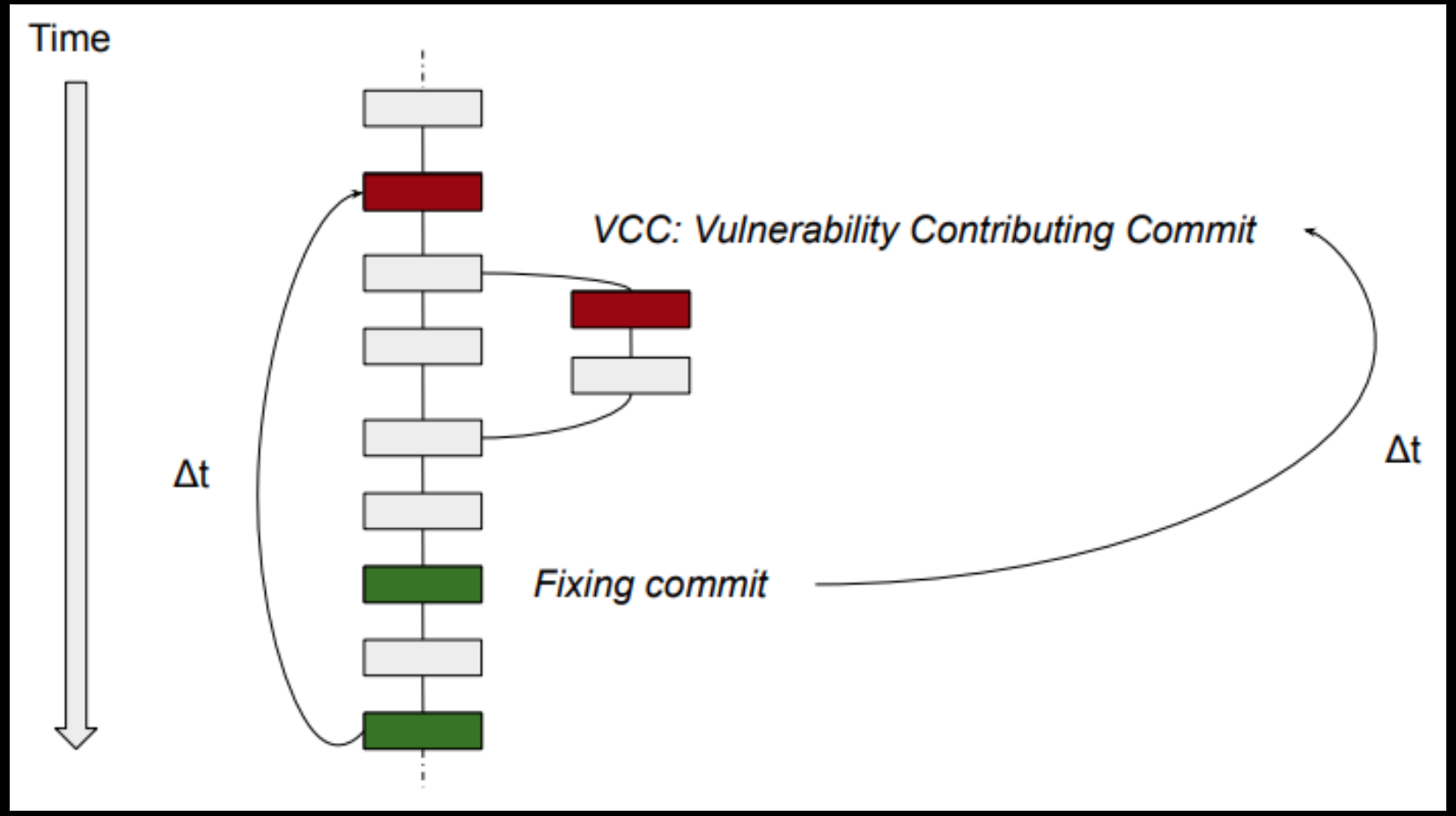
Figure 1: Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation.

alternatively the window of exposure to a vulnerability) begins with its introduction into a product (at time t_{int}). This first phase (Phase 1) of its lifecycle ends with its discovery by some party (t_f). Phase 2 covers the time period during which a vulnerability is known to at least one individual (and there is an associated risk of exploitation depending on their intentions), but is not publicly disclosed yet. Phase 3 begins with the public disclosure of the vulnerability (t_d) and ends

Vulnerability lifecycle



Code lifetime of vulnerability



A fixing commit, blamed

CVE-2022-25375

```
638  638          rndis_resp_t *r;
639  639                                                          Blames[7e27f18] += 1
     640  +       BufLength = le32_to_cpu(buf->InformationBufferLength);
     641  +       BufOffset = le32_to_cpu(buf->InformationBufferOffset);
     642  +       if ((BufLength > RNDIS_MAX_TOTAL_SIZE) ||
     643  +           (BufOffset + 8 >= RNDIS_MAX_TOTAL_SIZE))
     644  +                   return -EINVAL;
     645  +
640  646          r = rndis_add_response(params, sizeof(rndis_set_cmplt_type));   Blames[83210e5] += 1
641  647          if (!r)
642  648                  return -ENOMEM;
643  649          resp = (rndis_set_cmplt_type *)r->buf;
644  650
645       -      BufLength = le32_to_cpu(buf->InformationBufferLength);          Blames[a1df4e4] += 1
646       -      BufOffset = le32_to_cpu(buf->InformationBufferOffset);          Blames[a1df4e4] += 1
647       -                                                                      Blames[1da177e] += 1
648  651    #ifdef VERBOSE_DEBUG
```

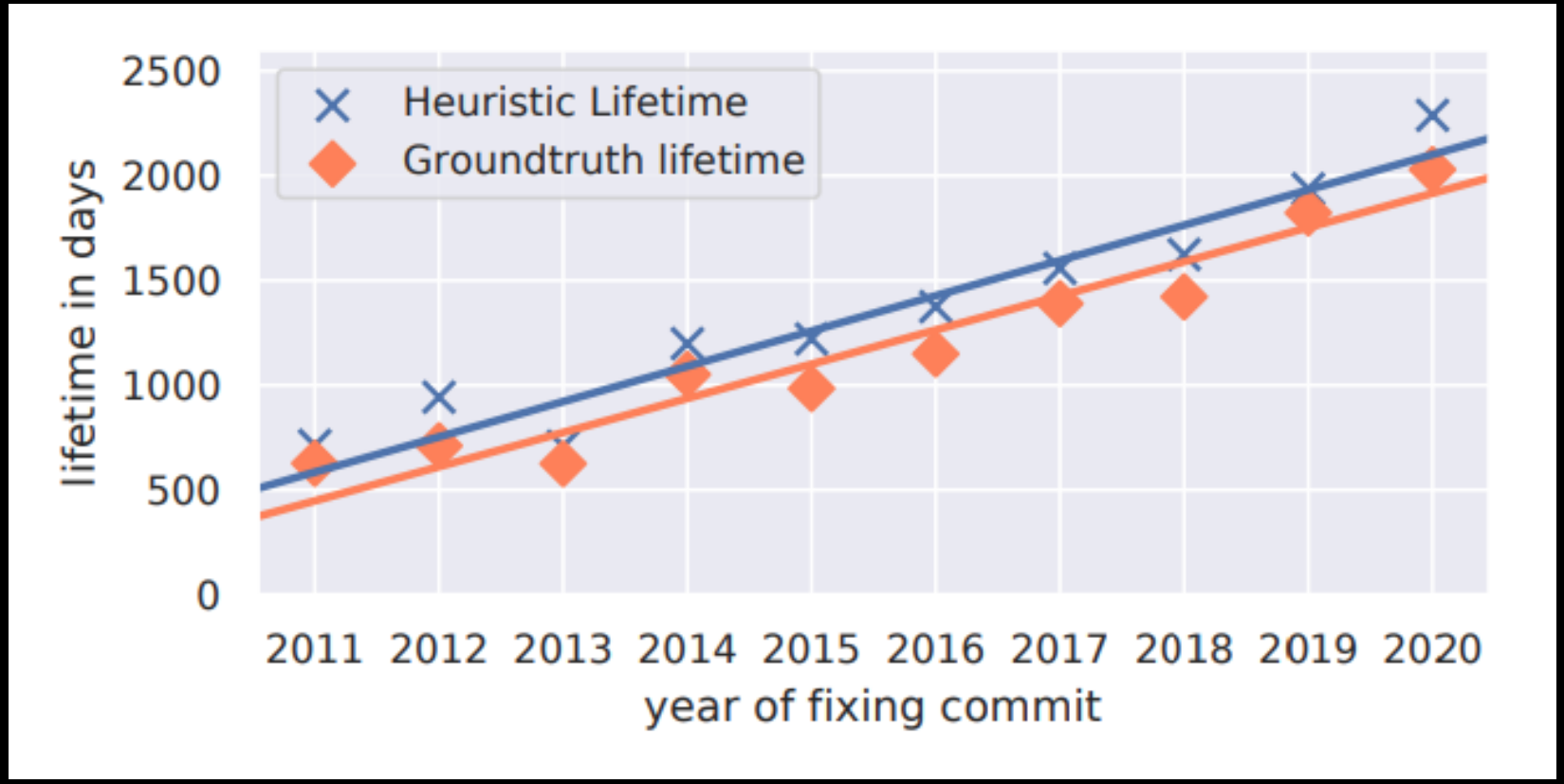
Estimating lifetimes

- Goal: approximate the point in time when a vulnerability was introduced – do not care about the actual VCC

$$d_h = d_{ref} + \frac{1}{\sum_{i=1}^{n} b_i} \sum_{i=1}^{n} b_i (d_i - d_{ref})$$

- Compute d_h of VCC as follows: Collect n commits based on the fix where b_i is # blames for the commit, and d_i is its date.
  - Blame every line that was removed
  - Blame before and after every added block of code (two or more lines) if it is not a function definition as these can be inserted arbitrarily.
  - Blame before and after each single line the fixing commit added if it contains at least one of these keywords ("if", "else", "goto", "return", "sizeof", "break", "NULL") or is a function call

Comparison to ground truth (Linux kernel)

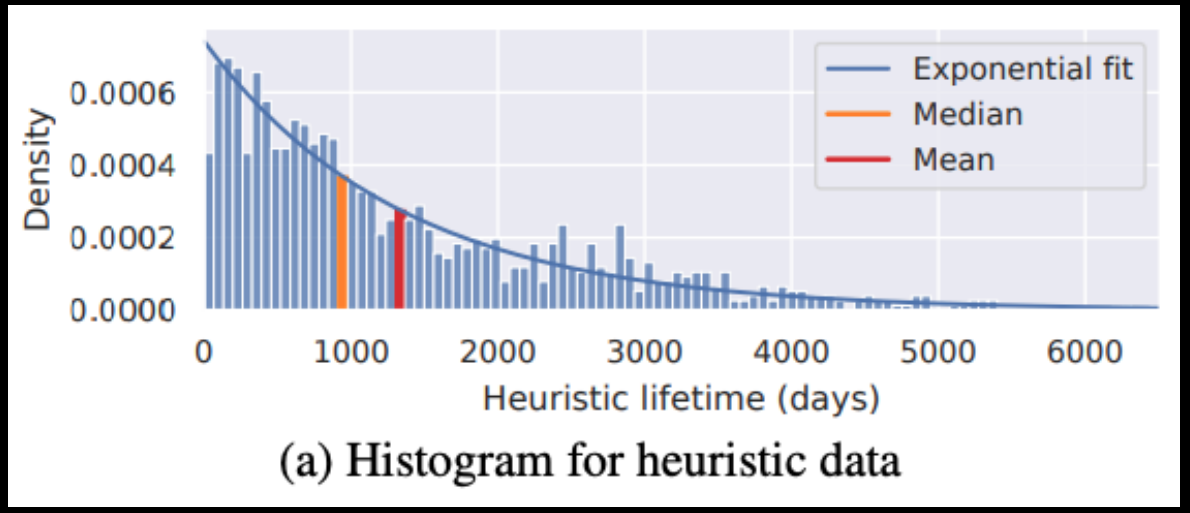


Study projects

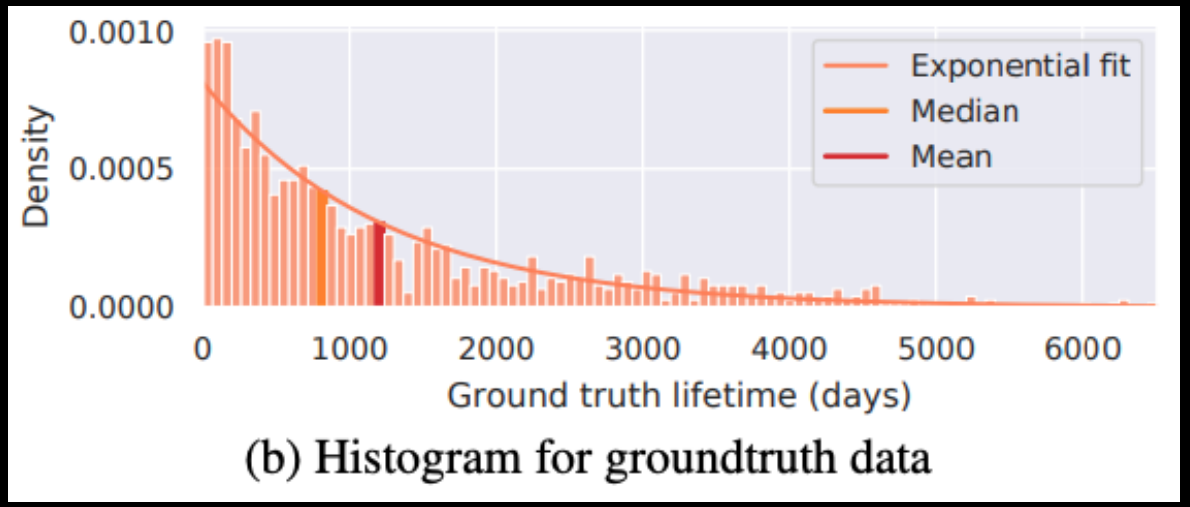
Project	CVEs	w/ fix. com.	# fix. com.
Linux (kernel)	4,302	1,473	1,528
Firefox	2,179	1,498	3,751
Chromium	2,781	1, 580	2,820
Wireshark	600	314	343
Php	663	281	932
Ffmpeg	326	277	373
Openssl	214	144	259
Httpd	248	132	476
Tcpdump	167	115	128
Qemu	340	213	290
Postgres	139	76	141
Total	**11,959**	**5,914**	**11,041**

Table 1: Number of CVEs and mappings per project. First column gives the total number of CVEs returned from a search of the NVD. Second column gives the number of those CVEs for which at least one fixing commit was found in the project repository. Third column gives the total number of fixing commits found per project.

Comparison against ground truth



(a) Histogram for heuristic data



(b) Histogram for groundtruth data

Average lifetime trend

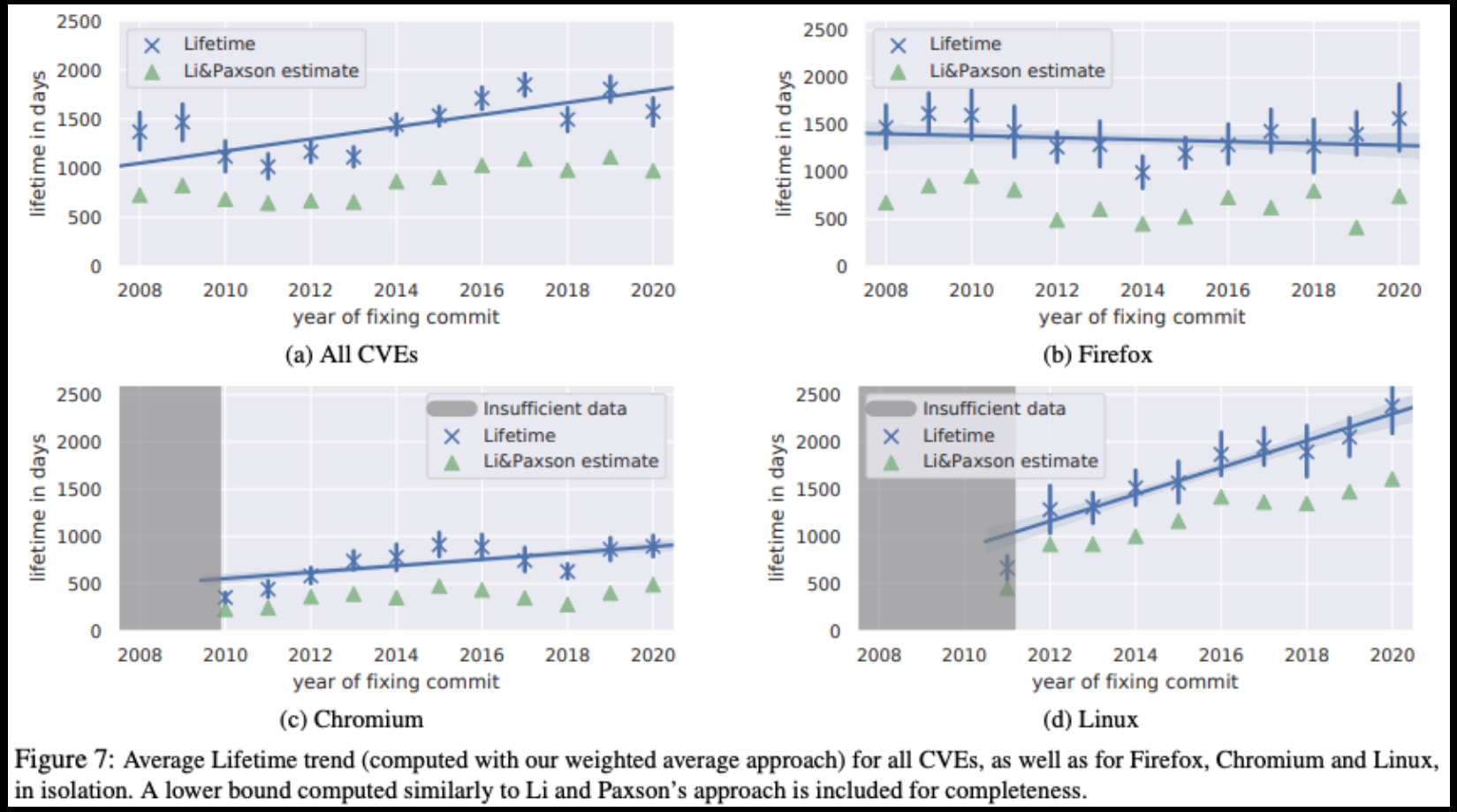


Figure 7: Average Lifetime trend (computed with our weighted average approach) for all CVEs, as well as for Firefox, Chromium and Linux, in isolation. A lower bound computed similarly to Li and Paxson's approach is included for completeness.

Lifetime trends vs. code age

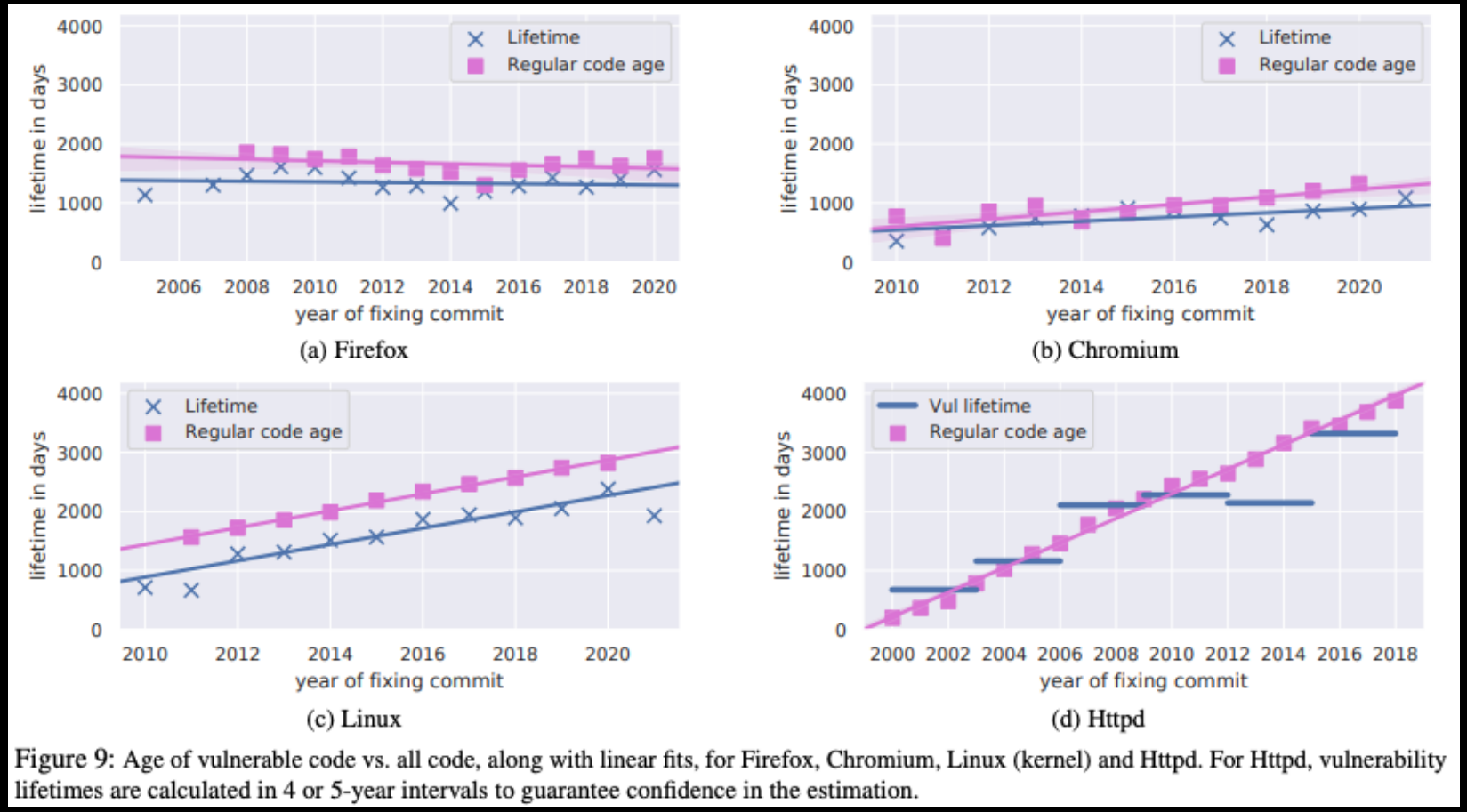


Figure 9: Age of vulnerable code vs. all code, along with linear fits, for Firefox, Chromium, Linux (kernel) and Httpd. For Httpd, vulnerability lifetimes are calculated in 4 or 5-year intervals to guarantee confidence in the estimation.

Surprise! No impact of fuzzing on Linux

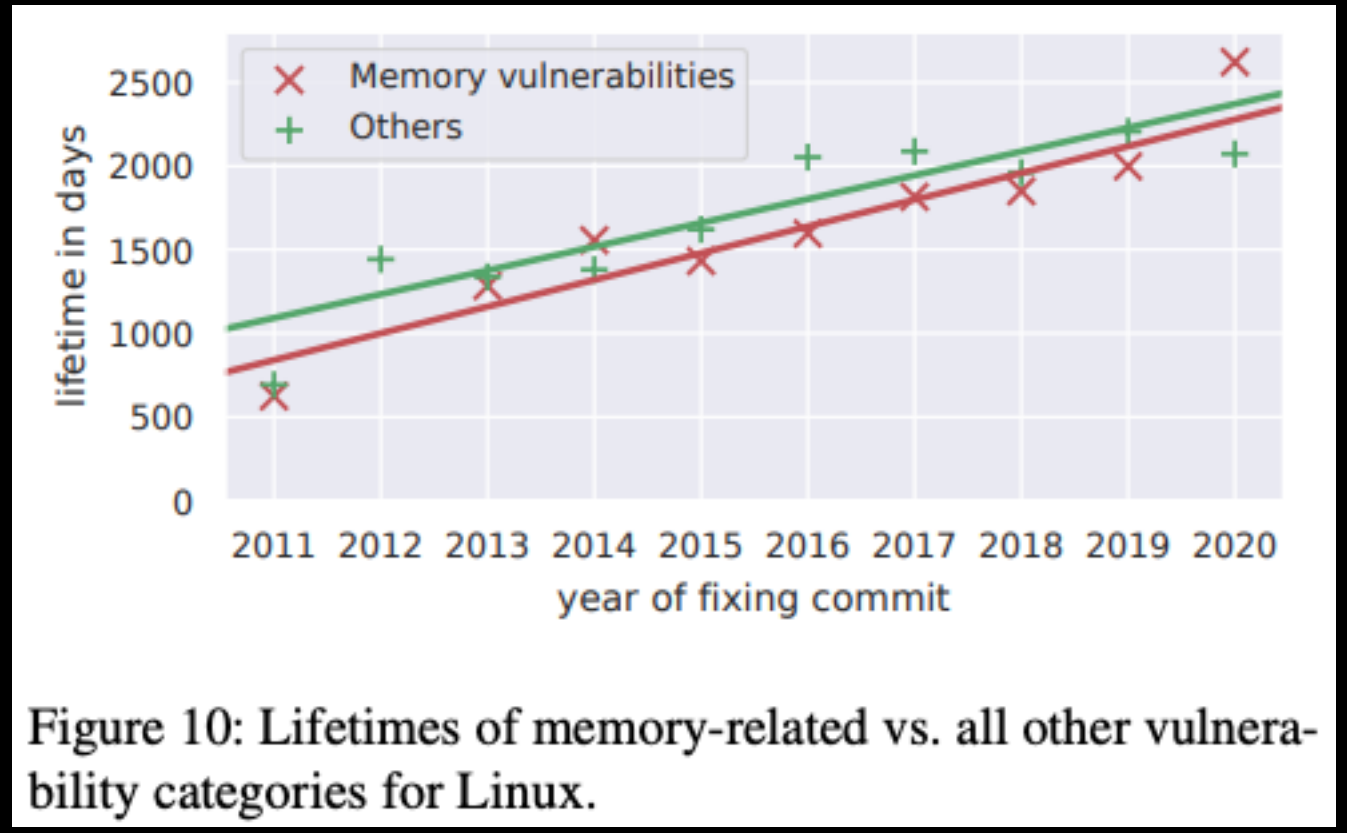


Figure 10: Lifetimes of memory-related vs. all other vulnerability categories for Linux.

Some analysis observations

- No evidence to support that we are introducing (and consequently fixing) significantly fewer new vulnerabilities over time.
- Different vulnerability categories seem to be equally difficult to find (at least post release)
- For some projects the vuln-age spread increases over time
  - Maybe this means that parts of the code are “mature” and have fewer vulns?
  - If so: We could be slowly progressing towards a state of relative maturity, where vulnerability lifetimes become stable over time and not correlated to code age, even if the latter is increasing
  - And: Fuzzing code that has recently changed is the best vulnerability discovery strategy. Yet: fuzzers keep discovering very old vulnerabilities