2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST) | 978-1-7281-6836-4/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICST49551.2021.00051

Applicable Micropatches and Where to Find Them: Finding and Applying New Security Hot Fixes to Old Software

Mac Malone
UNC Chapel Hill, USA
tydeu@cs.unc.edu

Yicheng Wang
UNC Chapel Hill, USA
yicheng@cs.unc.edu

Kevin Snow
Zeropoint Dynamics, USA
kevin@zeropointdynamics.com

Fabian Monrose
UNC Chapel Hill, USA
fabian@cs.unc.edu

***Abstract*—With the complexity and interdependency of modern software sharply rising, the impact of security vulnerabilities and thus the value of broadly available patches has increased drastically. Despite this, it is unclear if the current landscape supports the same level of patch discoverability as that of vulnerabilities — raising questions about whether patches are simply scare or if they are just hard to find (*i.e.,* are there a lot of "secret patches" [1]), and, equally important, what kind of patches are they (*e.g.,* are they micropatchable). We seek to assess the current state of patching by analyzing patches for a four-month period of recent Common Vulnerabilities and Exposures (CVEs). At first glance, the state of patching seems abysmal — only one-fourth of CVEs have a labelled patch on the National Vulnerability Database (NVD). However, by searching for indicators on other popular security trackers (*e.g.,* Debian's), we were able to find a lot more "secret patches", but the ratio of patched CVEs plateaued around fifty percent.**

Examining the discovered patches, we noticed that many were version updates, and less than one-tenth had machine accessible source-code micropatches. Using a custom tool that leverages contemporary version control, we were able to test the feasibility of automatically applying these micropatches to older versions of the software and found that approximately two-thirds of the patches can be applied to at least one old version. The failure cases were mostly due to lax practices pertaining to security fixes and general software development (*e.g.,* releasing the fix along with other extraneous features). Reflecting on our investigations, we surmise that between existence, discoverability, and versatility of security patches, existence and discoverability are the bigger problems. As to why this is the case, we find that the answer may lie in the perverse incentive structures of the industry. We conclude with possible remediations and hope that our work at least raises public awareness of the current state of patching and encourages future work to improve the situation.

***Index Terms*—micropatching; patch discovery; patch testing**

I. Introduction

Our world is growing increasingly reliant on computer technology in almost every way. As this technology becomes more sophisticated and commonplace, the tendency of software developers to rely on pre-existing libraries to accelerate development grows. A side-effect of this is that security vulnerabilities within common libraries can have far-reaching impacts. The intended way to mitigate this is the release and use of security patches. Unfortunately, many developers do not patch their applications, leaving them dependent on outdated libraries with known vulnerabilities [2]–[5].

However, it is not clear that the application developers are completely at fault. A lot of library patches are disseminated via version updates and therefore incur a high cost on the application developer to ensure that the software is still functional and correct after the update. This makes it difficult to then use the released patches in certain fields. For example, a substantial amount of IoT technology relies on specialized variants of existing software libraries satisfying the unique constraints of such platforms. In some enterprise settings, these devices may be so old that the developers of the embedded software are long gone and thus manual verification of the correctness of a complete update is difficult. Ideally, we would instead want micropatches, or changes that fix the security vulnerability without altering other semantics of the program, as they are much more easily verifiable. To this end, the U.S. Defense Advanced Research Projects Agency recently announced the Assured Micropatching (AMP) program [6], dedicated to advancing research in safely and automatically patching vulnerable devices in critical infrastructure.

Motivated by the need of expediently patching legacy code in mission critical systems, we set out to understand the feasibility of automatically integrating new patches into old versions of software libraries. Our approach is primarily developer-facing, taking the stance of someone dependent on a specific version of a library and seeing whether they can, upon learning of a vulnerability in the latest version of library, find a patch for it and automatically apply it to their own version.

We performed an evaluation of the feasibility of micropatching security vulnerabilities by asking several pertinent questions, including: (*RQ1*) whether disclosed vulnerabilities have publicly available patches; (*RQ2*) what percentage of the patches are machine-accessible micropatches that can be applied as is, *i.e.,* without relying on semantic-awareness or static analysis, and (*RQ3*) what factors may be related with whether patches are released or not. To answer these questions, we embark on an effort with a scope and approach that sets us apart from prior works on patch applicability [7], [8], and circumvents scalability issues inherit in such approaches. A summary of how we approached these research questions can be found in Figure 1.

Our findings were quite troubling at first glance — only about one-fourth of CVEs in the National Vulnerability

978-1-7281-6836-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ICST49551.2021.00051

Database (NVD) have listed patches, and there are a significant number of data entry errors in this reporting. However, we were able to find "secret patches" for many other CVEs by going beyond NVD and searching for indicators on other security trackers (*e.g.*,, those by Debian, SecurityFocus, and vendors like Apple). But, we experienced diminishing returns in our ability to find new patched CVEs with each additional datasource, leading us to conclude that at most half of all CVEs have publicly listed patches — an observation in line with modelling results conducted by MITRE [9]. However, almost half of these "patches" are software updates rather than actual patches, and are thus largely unusable by users who need guaranteed compatibility. The other half is split between direct links to patches and unstructured references (*e.g.,* mailing lists, bug trackers, or forums). While the latter may contain patches, their unstructured nature makes it hard to extract the patches automatically.

However, for the patches we could extract, the outlook is quite positive. We found that about two-thirds of available patches can be applied automatically using standard software management tools. Patches that cannot be applied are often due to lax policies regarding the release of security fixes — library maintainers often bundle such fixes together with major releases and other extraneous changes, making them hard to isolate and apply as a hot fix. Thus, even many of the patches we found are not true micropatches. Furthermore, we found that certain libraries release new versions infrequently. Thus, any infrastructure that relies on older versions of a library will suffer from known, severe vulnerabilities — potentially for years — despite the availability of existing patches.

Our results show that the primary threat to the feasibility of automated micropatching does not come from the difficulty of applying, building, testing, or verifying patches (which is already a Herculean effort in and of itself). Instead, it comes from acquiring micropatches for software vulnerabilities in the first place. We suspect that this dearth of publicly available micropatches is partially caused by the current incentive structure of patching software: *while there are bug bounties that provide incentive for the discovery of vulnerabilities, there is little to no immediate incentive for maintainers to patch those in a modular way.* Although there are incentives for 3rd-party service providers to bridge the gap between maintainers and users by creating their own micropatches,[1] it is also in their business interest to keep those patches private.

II. Approach

Our objective is to determine the general patchability of security vulnerabilities. For that, we require a database of security vulnerabilities to survey — ideally ones that would be representative or hold significant interest. As such, we looked to Common Vulnerabilities and Exposures (CVEs). A CVE is a public record of a cybersecurity vulnerability in a piece of software. These records are hosted by MITRE [10] and synchronized with the National Vulnerability Database (NVD)

[1]There are indeed companies that are providing this service, *e.g.,* Snyk.

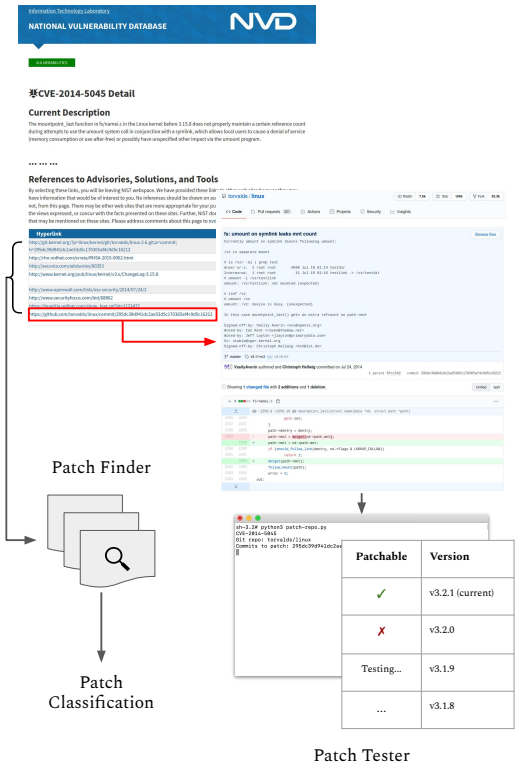

Fig. 1. A summary of our research approach: for each Common Vulnerability Enumeration (CVE), the National Vulnerability Database (NVD) maintains a list of publicly available references. We scan the references and classify them into different categories of accessibility (*RQ1*). When we encounter machine accessible micropatches (*e.g.,* those on github), we test if they can be applied as-is to different versions of the software (*RQ2*).

[11], which provides further details on the vulnerabilities reported. We leveraged `cve-search` [12] to obtain a copy of these online databases to speed up our analyses.

CVEs in NVD contain a number of external references containing details about the vulnerability. These references have tags which indicate what kind of material they link to. One type of reference is a "Patch". We thus hypothesized that a CVE, when patched, will contain a link to the patch in its references. We explore this hypothesis further in §III, where we discover other sources of patches.

Given such a *patch reference* in the form of an URL tied to a CVE, we assess what versions of the software it can likely be applied to. While the CVEs may provide information about the vulnerable versions of the software, they do not often provide us with information about which versions (if any) the *referenced patch* can be applied to. As such, we must test this ourselves. A side-effect of this is that, for our purposes, we require that both the software and its patch be open-sourced. We choose this heuristic as there is no general way to easily test the applicability of blackbox binary updates to previous versions of the software.

Open-source software is often managed by a version control system (VCS), which keeps track of changes (termed *commits*) made to the codebase (termed a *repository*) over its lifetime. A

TABLE I
CVSSV2 METRICS FOR SUBSETS OF THE CVE DATABASE (VALID CVES ONLY). DUE TO RATE-LIMITING FROM NVD, WE ESTIMATE RELEVANCE (*i.e.,* % OF CVES THAT REFERENCES A PATCH AND/OR EXPLOIT) BASED ON A RANDOM SAMPLE (10K) OF THE FULL DATABASE.

			Ease of Access			Relevance		
Dataset	**CVEs**	**Severe**	**No Auth**	**Low Complex**	**Net**	**Patched**	**Exploited**	**Both**
Full Database	136k	32.7%	82.0%	54.0%	80.0%	27.4%	23.4%	5.4%
In Time Period	6480	24.2%	80.4%	61.9%	80.3%	28.3%	27.1%	4.5%
With Commit	397	29.7%	89.7%	68.5%	87.7%	96.7%	22.7%	21.4%

VCS enables users to undo commits and revert the repository to a previous state (*i.e.,* an older version). To test if a patch applies to a given version of software, we load that version and see if we can apply the changes specified in the patch. Most patches are given in the form of a *diff*, which encodes information about modified files and specific line changes. If the file(s) and line(s) referenced in a diff have changed from those specified, the patch may be unapplicable (*e.g.,* you can't modify lines in a file that has since been deleted).

Our patch tester leverages one such version control system, Git, to test patches. While many other version control systems exist (*e.g.,*, Subversion and Mercurial), Git is the most widely used in open-source software projects [13], [14] and in CVEs [15]. For Git-backed software, a CVE generally links to commit(s) on a public Git repository (*i.e.,* URLs of the form `[repo]/commit/[hash]`), and the whole set of linked commits constitutes the patch. We use this information to download the repository and test the patch.

Given this data, we venture to provide insights on the security implications of the examined CVEs and corresponding vendor patches. For part of this assessment, we rely on the Common Vulnerability Scoring System Version 2 (CVSSv2) [16].[2] In CVSSv2, a score of 7 or above is considered "high" severity [16], which we use as our baseline for a "severe" CVE/patch. Additionally, CVSSv2 splits the security implications into two major categories: how the vulnerability is accessed, and the theoretical implications of exploiting it. It further splits each of these into three subcategories. For access, these are: the authentication level, the complexity of the access, and the vector through which the vulnerability can be accessed (*e.g.,* through the network or only locally on the machine). For impact, the standard security metrics of availability, confidentiality, and integrity are used. CVSSv2 denotes impact as none, partial, or complete to indicate how much the metric was compromised.

III. RQ1: WHERE DO ALL THE PATCHES GO?

For our analysis, we examined CVEs during a four month period covering the most recent CVEs at the time of our experiments. 7254 CVEs were published during the observation period, of which 774 were eventually rejected by NVD (*e.g.,* due to being duplicates), leaving us with 6480 valid CVEs (1542 of which were rated severe).

[2]While a newer version 3 has been released, `cve-search` uses version 2.

TABLE II
BREAKDOWN OF PATCHES FOUND

Version Controlled		
Git Commits	397	5.47%
GitHub Pull Requests	84	1.16%
Other	76	1.05%
Unstructured Discussions		
Bug Tracker	374	5.16%
Mailing Lists, Forums, etc.	66	0.91%
Update Only		
OSS Update	589	8.12%
Binary Update	1176	16.21%
Unclear	233	3.21%
No Patch Found	3485	48.04%
Rejected CVEs	774	10.67%

NVD annotates references with tags describing the link. For example, a reference may be tagged "Third Party Advisory", "Exploit", or, "Patch". Out of the valid CVEs within our timeframe, only 1834 (28.3%) had at least one reference link tagged as "Patch". Independently, 397 (5.5%) CVEs referenced a recognized Git commit and were thus considered "micropatches" testable by our approach (see Table II).

Not all the CVEs referencing Git commits were tagged as "Patch". While this could potentially be due to the commit being an exploit, or the change which introduced the vulnerability, a manual analysis confirmed 18 of the 19 untagged commits in our test set were patches. The one outlier was tagged "Release Notes" and was, in fact, release notes — the commit containing that patch was not included as a reference. For the other 18, the most common error was for the patch to be tagged "Third Party Advisory", though some were left untagged and 2 were mistakenly tagged "Exploit".

The small sizes of both the tagged patches and our testable micropatch subset raised many questions. First, were our test set, our time frame, and the CVE database at large similar? Second, why did we find so few patches? Third, was choosing Git too limiting or was there a larger issue, *i.e.,* are many CVEs just left unpatched?

To test for measurement bias, we compared the CVSSv2 security metrics along with the patch and exploit statistics[3] of our dataset with that of the larger sets. The results of this is shown in Table I. We found that our test set is reasonably similar to both the analyzed time frame and the

[3]What % of the CVEs referenced a tagged patch and exploit, respectively.

overall database. Looking more closely, we noted that the CVEs with patches and those with exploits are essentially disjoint. About one-fourth (27.4%) of the database has patches and one-fourth (23.4%) has exploits. But, only 5.4% has both. **This suggests that developers' decision to patch defects is based primarily on the nature of the codebase and their particular style rather than security concerns like the severity and exploitability of the vulnerability.**

We also noticed that even though our time frame is recent, the fraction of patched CVEs matches the full database. This implies that the patch rate of CVEs does not increase significantly with time (*i.e.,* a few month old CVE is just as likely to be patched as a decade old CVE). One possible reason for this is that many CVEs are published after a responsible disclosure period wherein the vendors patch the vulnerability (if they are going to). Previous work studying the lifecycle of CVEs supports this. For example, Li and Paxson [15] found 78.8% of vulnerabilities were patched before they were publicly disclosed and 26.4% of unpatched published CVEs remained so 30 days after publication.

A. Discovering Untagged Patches

Closer inspection of the listed reference links on NVD gave us many untagged patches. In particular, we found six sources that give structured information regarding if a patch exists that can be automatically scrapped, listed below.

- *support.apple.com*: Apple policy states that it does not discuss or confirm vulnerabilities until the necessary updates are available [17]. Thus, if a support.apple.com link is present in the CVEs list of references, we can deduce that the CVE has been patched.
- *security-tracker.debian.org/tracker*: The Debian Security Bug Tracker's website lists the status of the bug (*i.e.,* if it has been patched and in what versions). By parsing that information, we can identify if a patch has been released.
- *access.redhat.com*: RedHat has two sources we can use to find patches. RedHat Errata entries provide a list of "Fixes" that link patches to CVEs. RedHat customer portal reports directly on CVEs, providing a list of affected products and their versions.
- *bugzilla.redhat.com*: Each entry tags the bug's status.
- *www.securityfocus.com/bid*: Whether a fix is available for a CVE is noted under the "Solution" header.
- *usn.ubuntu.com*: If an update is available, there will be an "Update Instructions" section.

Furthermore, Debian Security Bug Tracker and Bugzilla provide mechanisms to search entries via CVE ID, so we can obtain patch information from those two sources even if links to those platforms are not referenced by NVD. Note that sources from several prominent vendors (*e.g.,* Microsoft, Oracle, Adobe, etc.) are not included in the above enumeration because those vendor's security updates are usually tagged as "Patch" on NVD. Leveraging these sources, we were able to automatically find patches for 1146 more CVEs, thereby significantly increasing the total number of CVEs with patches to 2979 — 46.0% of the analyzed time frame.

Unfortunately, evidence suggests that we may be nearing the upper bound of publicly available patches for CVEs. For one, in an extensive simulation on vulnerability management conducted by MITRE (using parameters that appear to be derived[4] from 24 years of data), Moore and Householder [9] note that "*even if the perception of the adequacy of vulnerability management on the part of the defenders was near perfect (i.e., if the defender were able to immediately patch any vulnerabilities they were aware of or could become aware of), the actual adequacy of vulnerability management still hovers below 0.5.*" This is corroborated by our analysis of how many new patches were found with the addition of each new data source. Figure 2 plots the cumulative total number of CVEs with patches as a function of number of additional data source referenced. We see that the data increases to an asymptote at 3016 — 46.5% of the CVEs in the time frame.

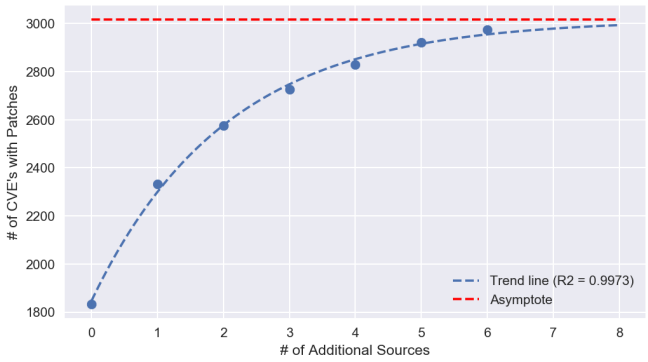

Fig. 2. The impact of using additional data sources to derive the number of CVEs with patches. The data suggests that only around 3000 out of 6480 valid CVEs have patches that exist in the wild.

B. Security Indicators & Patching Behavior

The low rate of CVEs with accessible patches motivated us to study potential causes for this type of behavior. We investigated whether the security implications of CVEs affects patch availability on NVD (explicit or hidden). To this end, we conduct a χ^2 homogeneity test to see if the distributions of CVEs with listed patches are different from the global distribution when controlled for various security parameters.

The results are presented in Table III. Overall, we are not able to find any clear trend from the results. However, a few data points are worth highlighting: we found that CVEs with a severe CVSSv2 score were less likely to have listed patches compared with the global distribution. We also found similar trends for both confidentiality and integrity impacts, whereby CVEs that have complete impacts in these areas were less likely to have listed patches. For access metrics, we were unable to reject the null hypothesis at $p < 0.05$ that the vector of access had any impact on if the CVE was patched. We did find, however, that CVEs with high access complexity were more likely to be patched compared the global distribution. We

[4]See, Allen Householder, *Analyzing 24 years of CVD*, at https://resources.sei.cmu.edu/asset_files/Presentation/2018_017_001_515355.pdf

TABLE III
IMPACT OF SECURITY METRICS ON A CVE'S PATCH AVAILABILITY. $\perp, +,$ AND $-$ MEANS "NO, POSITIVE, AND NEGATIVE STATISTICALLY SIGNIFICANT IMPACT" ($p < 0.05$), WHEN COMPARED TO THE GLOBAL DISTRIBUTION, RESPECTIVELY.

Metric		Value	Impact
Severity	CVSSv2 Score	≥ 7	$-$
		< 7	$\perp$
Ease of Access	Authentication	None	$+$
		Single	$-$
	Complexity	Low	$-$
		Med	$+$
		High	$+$
	Vector	Local	$\perp$
		Network	$\perp$
Security Impact	Availability	None	$-$
		Partial	$+$
		Complete	$\perp$
	Confidentiality	None	$\perp$
		Partial	$\perp$
		Complete	$-$
	Integrity	None	$+$
		Partial	$\perp$
		Complete	$-$

do not have firm hypotheses to explain these counter-intuitive findings, but they do demonstrate that security implications of CVEs are not correlated with patch availability in the ways one might expect or hope.

IV. RQ2: IS MICROPATCHING FEASIBLE?

In Git, we can reset a repository to the state it was in at any given commit, thus each commit can be thought of as a *version* of the repository. Git also allows us to apply the changes specified in a given set of commits (*i.e.,* a *patch*) to the current state of the repository through a process called *cherry-picking*. To test if a patch is applicable to a version of the codebase, we reset the codebase to the commit corresponding to that version and cherry-pick the set of commits corresponding to the patch. Note that this approach tests applicability at a course granularity — even if the patch can be applied in Git, it could still lead to compilation or logical errors. But, this approach does offer an *upper bound* to the number of easily applicable patches there are, and is important for studying the feasibility of applying existing patches as micropatches.

While we could do this for every commit, repositories can have hundreds of thousands of commits. Thus, automatically checking if each of these commits is patchable can be extremely time-consuming. As such, we sought a way of narrowing down the commits we treat as versions. Git provides a feature called tags, which allows the user to denote significant commits with a label. Many open-source projects use said tags to denote different versions of the software (*e.g.,* the Linux GitHub repository [18]). While some libraries do not use tags for this purpose, we hypothesize that they will nonetheless generally serve as a good proxy for versions.

A. Analysis

To test the general feasibility of applying identified patches, we ran the 397 CVEs we found with recognized Git patches through our patcher. 9 CVEs had two different patches (*e.g.,* due to the software being mirrored across different repositories), making for a total of 406 patches (101 with a severe rating). These patches were spread across 185 repositories, some of which have versions dating back to early 2000. Table IV provides data for 15 of the most patched repositories within our timeframe.

The "Versions" column gives the total number of tagged releases within the repository (covering its entire lifetime) and the average (mean) number of these releases a patch can be applied to. The "Patches" columns shows the total number of patches in the test set for each repository and the number of those patches that were not applicable. For each of these categories, we also provide, in parentheses, the number of patches that were classified as severe.

The "Popularity" column lists install statistics for the software from Linux and PHP package managers and the number of stars and forks on the GitHub repository. A dash indicates that no install data was reported for the software, and a diamond ($\diamond$) indicates that the number reported was much less than the star count, implying that number was likely not representative of the software's popularity. For example, C libraries (*e.g.,* libyang, libpcap) are generally not installed, but rather downloaded and linked to dependent software.

Lastly, the "Security Implications" summarizes the impact of the most severe CVE for each repository. A checkmark in the "Net" column indicates that the vulnerability is exploitable over the network. For availability, confidentiality, and integrity columns, the circle indicates how much of that aspect of security was compromised. A full circle indicates that the vulnerability completely compromised that aspect, a partial circle indicates a partial compromise, and an empty circle represents no compromise.

The table shows, for example, that for SQLite and WordPress, we were unable to apply *any* of the patches for these libraries. This is due to the fact that all the patches for these libraries contain extraneous version metadata that changes every commit. The table also indicates that, for FusionPBX, we found extraordinarily few versions. This is because it is one of the libraries that releases new versions very infrequently. For example, despite very active development over the past 2 years (over 2.6k new commits), it has not released a new version. We provide a more in-depth discussion of results like these later in Section IV-B.

We also performed a Latent Dirichlet Allocation (LDA) topic analysis on the summaries of the accessible CVEs and found 5 topics that cover a wide range of vulnerabilities.[5] The description and keywords for the topics are given in Table V. Overall, the data is a representative subset of the full CVE database as we cannot reject the null hypothesis that the

[5]We selected the optimal topic number via the elbow method on the coherence of the LDA topics.

TABLE IV
STATISTICS FOR 15 OF THE TOP REPOSITORIES

		Versions		Patches (Severe)		Popularity			Security Implications			
Repository	**Application (Lang)**	**Total**	**Avg**	**Tested**	**Not App**	**Installs**	**Stars**	**Forks**	**Net**	**Avl**	**Con**	**Int**
Linux	OS (C)	650	123	59 (31)	3 (1)	◇	91k	31.7k	✓	●	○	○
FusionPBX	Telephony (Web/PHP)	6	2	33 (3)	19 (1)	-	399	429	✓	●	●	●
TCPdump	Networking (C)	42	3	22 (0)	12 (0)	2829k	1.3k	564	✓	◐	◐	◐
SQLite	Database (C)	114	0	13 (13)	13 (3)	600k	413	65	✓	◐	◐	◐
libyang	YANG Parser (C)	18	12	10 (2)	0	◇	199	147	✓	◐	◐	◐
VLC	Media (C)	50	3	6 (0)	3 (0)	1336k	6.1k	2.3k	✓	◐	◐	◐
libpcap	Networking (C)	45	1	5 (0)	3 (0)	◇	1.3k	536	✓	○	○	◐
WordPress	Publishing (Web/PHP)	410	0	5 (2)	5 (2)	◇	13.8k	8.5k	✓	◐	◐	◐
Pimcore	Publishing (Web/PHP)	113	35	5 (1)	0	290k	1.8k	835	✓	◐	◐	◐
Opencast	Video (Web/Java)	141	36	5 (1)	0	-	141	118	✓	◐	◐	◐
QEMU	Emulator (C)	276	117	4 (0)	0	102k	3.6k	2.8k		●	●	●
HHVM	Hack Interpreter (C++)	722	504	4 (4)	0	◇	16.5k	2.9k	✓	◐	◐	◐
WordPress Dev.	Publishing (Web/PHP)	410	28	4 (1)	0	-	356	408	✓	◐	◐	◐
Python Pillow	Imaging (Python/C)	65	4	4 (0)	0	-	7.4k	1.5k	✓	◐	◐	◐
ImageMagick	Imaging (C)	214	35	3 (0)	0	1563k	3.9k	641	✓	◐	◐	◐

distribution of "severe" CVEs amongst the 397 tested CVEs is different from that of the full database at $p < 0.05$ (done via a chi-squared test of homogeneity).

Additionally, an analysis of the primary programming languages in the 185 repositories (see Figure 3) shows a large variety of languages, providing evidence that our language agnostic approach to patch testing allowed us to analyze a much wider subset than a language-specific semantic-aware approach would have permitted.

TABLE V
DOMINANT TOPICS (WITH CONTRIBUTING KEYWORDS FROM LDA ANALYSIS) WITHIN THE SUMMARIES OF THE TESTED CVES.

Topic	**%**	**Keywords**
Code Injection	23.43%	arbitrary, craft, input, execute
Privelege Management	22.67%	user, access, privilege, file
Memory Management	21.41%	memory, service, function, leak
Various Binary Exploits	19.90%	code execution, stack, issue
Web Traffic Parsing	12.6%	request, header, attack, http

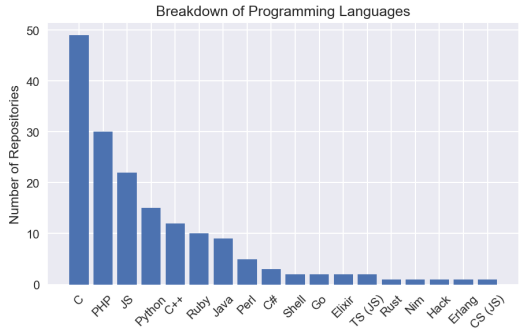

Fig. 3. Breakdown of primary programming languages in the analyzed repositories. Note that 78% of the repositories have a primary language that makes up greater than 80% of the codebase.

We also measured how far back in time a given patch could be applied and grouped the data by severity and repository. The results of this analysis for the 279 applicable patches can be seen in Figure 4. The mean and standard deviation was 1.6/2.3 years for all patches and 1.3/2.0 years for severe patches. When grouped by repository, the mean and standard deviation was 1.8/2.5 years and 1.5/2.3 years for all and severe patches, respectively.

Summary of Findings: Among the 406 patches, about two-thirds (279 total; 78 severe) can be applied to at least one version of the repository. This is positive news, indicating the feasibility of this direction of research. On average, a patch can be applied to 34 versions of a repository with a standard deviation of 1.3. The other 127 patches (23 severe) could not be applied to any version. Similarly, about one-third (60) of the 185 tested repositories contained at least one unapplicable patch. Next, we take a closer look at these failures.

B. Understanding Why Patching Fails

The 127 patches that could not be applied to any version fell into 6 categories (see Table VI). In what follows, we examine four major points of failures.

TABLE VI
SUMMARY OF PATCH CATEGORIES

		Security Implications			
Category	**Patches (Severe)**	**Net**	**Avl**	**Con**	**Int**
Ill-formed	11 (3)	✓	●	◐	◐
No Tags	10 (5)	✓	●	●	●
Alpha Patch	3 (1)	✓	●	●	●
Not Yet Released	29 (2)	✓	●	●	●
Multiple Commits	20 (6)	✓	●	◐	◐
General Conflicts	54 (6)	✓	●	●	●
Unapplicable	**127 (23)**	✓	●	●	●
Applicable	**279 (78)**	✓	●	●	●

The first issue was that some patches were ill-formed. Attempting to cherry-pick the commit referenced in the patch was unsuccessful as the version control system deemed the commit invalid. The second complication was that some

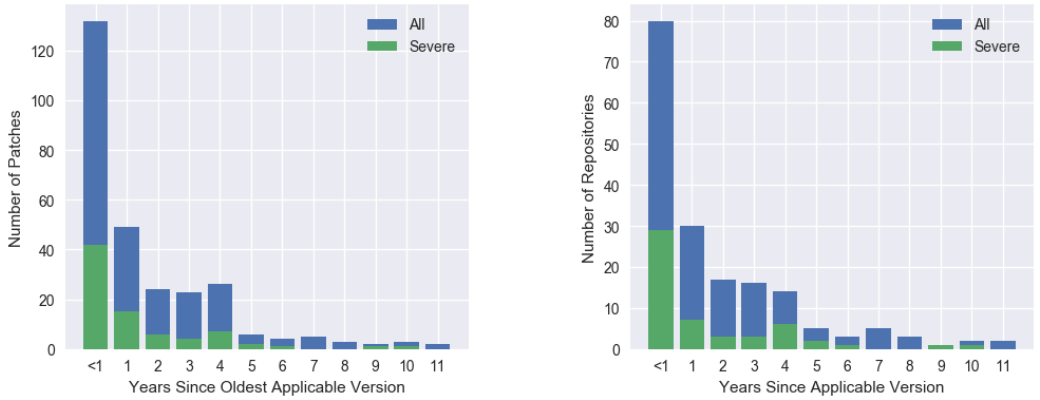

Fig. 4. Backward Compatibility of Patches

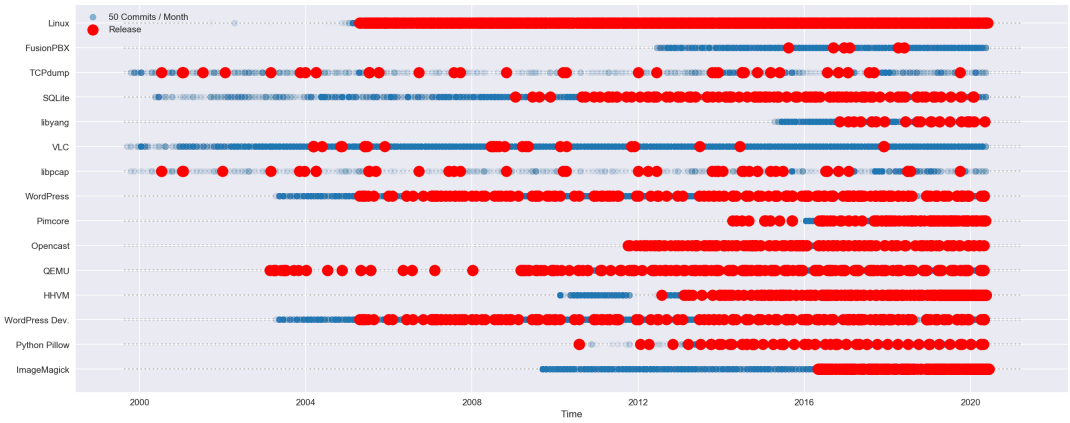

Fig. 5. Timeline of the commits and releases of the top repositories.

repositories contained insufficient version information. For instance, some repositories simply do not mark versions at all or use branches instead of tags. Other repositories release versions infrequently or have just begun tagging them. Patches in the "Alpha Patch" category were made before versions of the software were tagged (*i.e.,* they were made when the library was in "alpha" state before official releases). Thus, changes between the patch and first formal release made these impossible to apply. Patches in the "Not Yet Released" category were made after the most recent released version and thus do not have a release containing them. Thus, changes between the last tagged version and the patch made these impossible to apply.

To illustrate these issues, Figure 5 depicts the commits and releases of the 15 repositories in Table IV. Notice that, for example, despite active development, FusionPBX has not released a new version in about 2 years. Hence, it is very unlikely that a new patch will apply to any version of the software (especially considering the average number of years back a patch applies is 1.6). Situations like these explain why "Not Yet Released" patches is the predominant source of failure for the patch tester.

The third problem was patches with multiple commits. While Git does allow cherry-picking multiple commits at once, it has a number of restrictions on what kinds of commits can be applied simultaneously. For example, one can not cherry-pick both a merge and regular commit. These limitations are often too restrictive and make it impossible to apply a patch.

Another point of failure was with the patches themselves. Some patches contained extraneous changes in addition to the main security patch. Changes to metadata such as comments, change logs, or tests produced conflicts that made it impossible to apply the patch. For example, WordPress has a file with a version string that changes each build and SQLite keeps a manifest file in its repository containing hashes of each file, thus every commit changes these files and produces conflicts. Other patches were modified before the next release (*e.g.,* to fix a bug in the original patch, or better incorporate it into the structure of the codebase). Patches were also incorporated into major and minor version releases rather than being released independently as a fix. All of this served to make these patches unapplicable to both previous and subsequent versions.

V. Expanding the Hunt

Despite the promising success rate of direct applicability, it's important to keep in mind that only a small amount of CVEs had patches accessible to our tester. We therefore attempted to expand the set in two ways: by testing links to pull requests and searching for commits within the unstructured references. We also present some preliminary work in fabricating micropatches by obtaining the vulnerable and then the subsequent fixed version of the codebase to create a diff.

A. Pull Requests

Version control services (*e.g.,* GitHub, GitLab) provide a formal way of asking developers to merge changes into the main repository. The patch developer forks the stock repository, makes the necessary changes to the fork, and then creates a *pull request* that asks the owners of the repository to merge the changes from the fork. The developers can then approve the pull request, and the VCS will automatically perform the merge required to incorporate the given changes.

Given that understanding, we searched the valid CVEs in our time frame for references to GitHub pull requests and found 87 CVEs that listed such a link. We modified our patch tester to run on these links by leveraging GitHub's API to analyze the pull request and extract the commits contained within them. We excluded the 10 CVEs with closed pull requests, as that means the developer rejected the changes and therefore the patch may be invalid (or, alternatively, as we sometimes observed, the developer just does not care enough to incorporate the patch).

Of the 77 remaining CVEs, 67 were new – the other 10 were also in the initial test set (*i.e.,* they also contained direct references to commits). The 67 CVEs had 67 patches, of which 41 were applicable and 26 where not. In the initial approach, the predominant issue with patches was the patches themselves (*e.g.,* the patch was modified or incorporated extraneous changes). However, in this analysis, 23 of the 26 unapplicable patches were the result of metadata errors.

In summary, pull requests are more likely to contain patches which, if valid, are applicable. This makes sense, as pull requests are generally focused on a particular type of change that the requester wants merged, whereas commits made haphazardly to the main repository may combine many types of changes. This is one reason why forcing all development on a project to occur through pull requests is now considered best practice for Git — what is known as pull-based development or the fork and pull model [19].

B. Links within References

Table II showed that 6% of the patches have links to unstructured discussion forums (*e.g.,* bug trackers and mailing lists). Such forums could contain links to the commit(s) constituting the patch nested somewhere within them. To attempt to capture some of these patches in our analysis, we developed an extension that looks for links to commits *within* the references listed on NVD.

This search found 103 Git patches for 103 new CVEs. Of these, 62 were applicable. Of the 41 unapplicable patches, 27 were due to invalid commits. The high quantity of invalid commits leads us to believe that there may not be a strong connection between the links we found and actual patches for the CVE. Such references could easily be to the commit that introduced the bug rather than the one that fixed it or to a potential idea for a patch that was later discarded. This is compounded by the fact our search found a lot of non-commit links (*e.g.,* links to the main branch of a repository), demonstrating that users do, in fact, include version control system links to things other than patches within these references.

C. Source Code for Version Updates

Lastly, we explored obtaining the source code for vulnerable and patched versions of the software, and constructing a patch by calculating the cumulative diff between these versions.

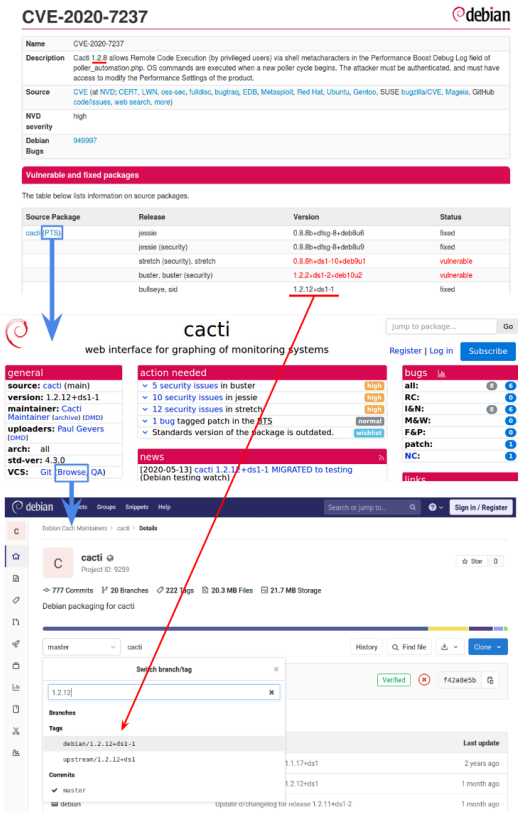

Fig. 6. Extract the entry by querying the CVE_ID, retrieve the affected and fixed versions (underlined in red), and retrieve the source code via links within the web page (highlighted in blue).

We note that while recent work by Dong et al. [20] provided an impressive machine learning approach to extract vulnerable software versions from unstructured texts, we leverage existing structure to circumvent the need for such heavy machinery. Specifically, we focused on the Debian Security Bug Tracker because it has several desirable traits: (i) Unlike other security trackers, the Debian Security Tracker exists within the Debian

eco-system and is crucially linked to codebases maintained by Debian. (ii) Debian Security Trackers presents vulnerable / fixed versions as structured first-class data within their web UI, allowing for easy access both by human and machines.

Specifically, we obtain the affected version by using a regular expression that recognizes semantic version numbers [21] and extracts all such numbers mentioned in the description of the CVE.[6] This set of candidate affected versions is noisy, as the CVE description could contain versions for different affected softwares or use a versioning system different from the semantic versions used by Debian, or even reference the fixed versions [20]. We filter this set of candidates by taking only versions that precede those marked as "fixed" by Debian. A link to the affected program's version control system (Debian's GitLab Server, salsa.debian.org) can be obtained by the process shown in Figure 6. Both the affected and fixed versions are tags for the Git Repository, which allows us to obtain source code differences that contain the security patch.

We believe that performing the cross-reference is sufficient to ensure accuracy for two reasons. Firstly, Debian is the primary source for their version of the software, and is more trustworthy than other sources that simply aggregate security information (*e.g.,*, NVD). Secondly, Dong et al. [20] showed that while most websites fail to maintain a complete list of vulnerable versions, in over 80% of the cases, the list they do maintain is correct but incomplete. This is not a problem for us, because all we need is a single vulnerable version.

Through this method, we were able to find accessible source code patches for 135 new CVEs, *including ones that did not list any patches at all on NVD*. While this method is significantly more crude than the other two in that it does not point to a commit that fixes the issue, it is an exploratory step that can be improved in the future. For example, we could scan through and analyze all the commits between the two tags to find those we recognize as security fixes (*e.g.,* by a reference to an issue or to a CVE ID).

VI. RQ3: Incentives for Secret Patches?

Even after our extensive efforts, we were not widely successful in finding bug fixes. To us, it was baffling that there was a deluge of vulnerabilities being disclosed, yet a lack of available patches. Given the rapid increase in the number of disclosed vulnerabilities in the past few years, coupled with the potential for economic damages to end users, one would expect that understanding the factors that contribute to the timing of vendors' release of patches would be a top priority. Indeed, reflecting on the findings from our study, it begs the question: *why are patches not more readily available?* From what we discovered, the reasons are complex, and several factors appear to influence whether vendors make patches available.

The most obvious factor is that getting a fix out can be time consuming, especially in large code bases. The patches themselves must be subject to the same development, design, and complexity challenges as any other piece of software. The fix may not be easy, and developers need to ensure that patches do not cause software regression (*e.g.,* performance degradation or instability), or worse, introduce new vulnerabilities. More subtle is the fact that market forces appear to be at play. For example, Arora et al. [22] showed that market size has a crucial role on a vendor's patching behavior. Interestingly, concentration in many software markets, indirect competition and threat of disclosure from vendors in complementary markets, can help reduce patching times almost as much as increases in the number of direct competitors. Jo [23] also found that even when software is provided "free of charge" to users (as in the browser space), companies still have incentives to fix known flaws even in the absence of direct competitors because they derive their revenues from neighboring markets (*e.g.,* from advertisers or firms that that buy usage traffic).

Conversely, the lack of competition has an adverse impact on not only developers' motivation to write secure code but also the way vendors respond to vulnerabilities discovered in their products [22]. This is likely due to the fact that in software market segments with a dominant vendor, software users have weak bargaining power resulting in minimal influence on a vendor's patch release behavior.

To complicate matters even further, perverse incentives can directly impact the release of patches. On one hand, the world of hacking can be viewed as a market: buyers seek the best price and sellers try to make the most profit from their discoveries [24], [25]. Arguably, the presence of markets can make activities more efficient, regardless of whether the activities it supports are laudable or not. Take, for instance, bug bounty and vulnerability reward markets. Even if we ignore the ethical concerns associated with such markets, the fact that these markets are unregulated can lead to problems that can not be overlooked. These markets provide financial incentives to individuals and organizations to discover and sell vulnerabilities, which result in more discovered vulnerabilities. That in itself is not problematic as it may motivate software developers to come up with relevant patches, and it also helps alert users about vulnerable software. But, the discovery and disclosure of vulnerabilities can directly translate into more attacks against vulnerable systems, and the time to appearance of exploits is shortening [26], [27]. Furthermore, data show that the volume of disclosed vulnerabilities has skyrocketed in recent years (see Figure 7), likely due to advances in both technology (*e.g.,* fuzzing) and market incentives. The deluge has overloaded the current infrastructure, leading to worse data quality and a smaller number of publicly available patches. So we ask, *if a large fraction of patches are not readily available, do vulnerability disclosures really improve the state of cybersecurity when published vulnerabilities without patches get quickly exploited [28]?*

In pursuit of that exact question, we realized that the wide gap between the number of disclosed vulnerabilities and available patches has not gone unnoticed. Of late, several

[6]The descriptions on the Debian Security Bug Tracker mirrors NVD's.

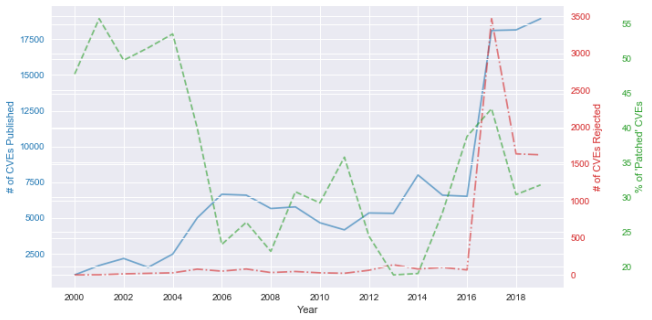

Fig. 7. NVD data throughput between 2000-2020.

commercial security vendors[7] offer services that attempt to fill the void, claiming to keep their customers' software manifests up to date even when patches for known vulnerabilities are not available on disclosure portals like NVD. But this is a slippery slope, as it could be financially rewarding to spend effort hunting for — and subsequently announcing — vulnerabilities in open source software, yet keeping patches private, only to have users of those affected packages pay a premium to stay up-to-date. Such tactics are not uncommon with vulnerability discoverers who seek rewards for their capabilities. For instance, in a study conducted by Algarni and Malaiya [29] of the most successful vulnerability discoverers, one top discoverer exclaimed that "*the main reason he became a vulnerability discoverer is that it made his own website more popular and enabled him to offer a source code review service*," while another declared "*he believes that the most profitable option for a vulnerability discoverer like him is to offer software security auditing services.*"

Lastly, even with renewed calls for stricter software security policy [30] and sanctions [31], vendors may still not have enough incentive to invest the time and effort to release patches. While multiple studies have shown that the announcement that a company experienced a security breach has a negative impact on its stock price, the long-term effects are less clear [32]–[34], but shows signs of being on the decline. In particular, Gordon et al. [35] suggest that "*there seems to be a shift in attitude among investors toward viewing information security breaches as creating a corporate nuisance rather than creating a potentially serious economic threat to the survival of firms.*" Moreover, the software that enabled the breach is rarely at the forefront of news stories, and so in the absence of a strong disapproval from investors, software producers are more likely to continue doing business as usual.

Takeaway: Both the emergence of a private vulnerability market place and the lack of incentive to publish patches are troubling trends. Although it is not entirely clear how to resolve the issue completely, it is clear from the data that current initiatives (*e.g.,* those run by MITRE) are ill-equipped to deal with the situation (see Figure 7). Moving forward, new efforts are called for that make the release of patches a first-class priority on par with vulnerabilities.

[7]See, for example, *The State of Open Source Security Report* by Snyk and *2020 Open Source Security and Risk Analysis (OSSRA) Report* by Synopsys

VII. Related Work

Patch behavior: Conventional wisdom suggests that users tend to delay software updates for fear of workflow disruption: an update might force them to learn new methods of completing tasks they had already mastered [36]–[41]. Generally speaking, because users are not provided with enough information to make informed decisions about whether to upgrade, the perceived marginal costs are high. Bergman and Whittaker [39] argued that postponing updates is rational because users must make decisions in the absence of critical information about hypothetical benefits, which is hard to do without actual experience of the new features.

Users also learn from negative experiences and delay patches even if the risks of not patching might be high [4], [5], [38]–[40], [42]. On the other hand, frequent patching may not protect a user from attacks if new vulnerabilities are frequently introduced into the codebase [43]. The frequency of updates can also impact adoption rate. Indeed, users rarely update within the first month of a patch's release and widespread adoption takes much longer [43], [44]. The tale for expert users is not much different; they only install updates 1.5 times more often than non-experts [45], and even system administrators routinely evaluate the pros and cons before apprehensively deciding to apply patches [46].

Patch analysis: Over the past decade, there has been an abundance of research on categorizing patches to investigate trends [47], [48], find similarities between patched code [45], examine whether a patch fixes a bug or offers an enhancement [15], [49]–[53], and determine whether a patch is in fact safe to apply [8]. Like us, many of these works only use the information provided by the patch itself for categorization and do not delve into semantics.

That said, this information (*e.g.,* commit diffs) generally lacks the expressiveness required to reason *meaningfully* about the nature of the patch [54]–[56]. Thus, several approaches augment patch analysis with static analysis or symbolic execution of the code (*e.g.,* [57]–[63]). Unfortunately, these techniques suffer from scalability issues and often restrict the analysis to a small set of programming languages or types of changes. We forego the application of more in-depth approaches as our goal is not to analyze the nature of the patch, but only to test if it can be applied.

Automatic patching: Recent research has studied methods of automatically updating software libraries. Specifically, OSSPatcher [7] attempts to apply patches for Android applications at runtime without involving developers. OSSPatcher takes a source code patch of an open-source library and generates a binary patch which can be applied at run-time via static analysis and reverse engineered build configurations.

While the technique used by OSSPatcher can be successful in applying hot fixes, it has practical limitations. First, the use of static analysis constrains evaluation to a small subset of languages (*e.g.,* C/C++, Java) and second, the build stage

constrains the technique to a small set of environments. Lastly, to make the approach scalable, simplifying assumptions must be made, further constraining which types of patches can be analyzed (*e.g.,* not supporting patches that modify types, but only those which add or remove them). In contrast, we avoid such limiting assumptions by studying the problem of how many patches can be applied verbatim at the source code level.

VIII. Conclusion & Future Work

We performed a large-scale feasibility analysis of the micropatching space and discovered that the manner in which patches are released leaves much to be desired. On the National Vulnerability Database, the gold standard of vulnerability disclosure, only one-fourth of the vulnerabilities are reported patched. On the bright side, this deficit can be improved somewhat by finding "secret patches" not reported on NVD but discoverable through other bug trackers, with the number of patched vulnerabilities likely plateauing at about 50%. Unfortunately, the patches themselves are often disclosed in a manner that is not machine accessible and many supposed patches are just software updates.

However, of the less than 10% of vulnerabilities with extractable source code patches, about two-thirds of these patches can be readily applied with standard VCS tooling. Furthermore, the other one-third failed primarily due to the nature of the patch —- developers often bundle security fixes with other major changes. This makes them unamendable to micropatching and means applications may not be able to incorporate them. Thus, if more of these patches were true micropatches, the number of applicable patches would likely be much higher. This further demonstrates that the major threat to the feasibility of micropatching is not the difficulty in applying and verifying patches, but rather the lack of available micropatches. We also discovered that core software libraries often release new versions infrequently, meaning patches are rarely timely. Thus, infrastructure that relies on specific stock versions of libraries will often continue to suffer from severe vulnerabilities even when (and if) they are reported patched.

Digging deeper, we discovered that the security industry has a strong incentive to publicize vulnerabilities, but little incentive to construct or publicize micropatches for them. We hope our discoveries highlight some important contextual issues within the area of automated patching, and spur the development of better practices when it comes to the creation and reporting of security patches.

Moving forward, we recommend that practitioners explore opportunities for autonomous cyber reasoning systems that incorporate techniques for accurately identifying source code changes that actually represent security patches. One intriguing direction is to systematically monitor source code repositories to detect silent fixes [1], [64], [65] or secret patches that vendors address without creating CVE entries or explicitly labeling the corresponding modifications in change logs [66], [67]. The main challenge is how to reliably relate vulnerability reports with the code changes in the vendors' repositories that provide the fix — and to do so at scale.

Acknowledgements

We thank the anonymous reviewers for their suggestions on how to improve the paper. We also express our gratitude to Murray Anderegg for his assistance with deploying the infrastructure used in this study. This work was supported in part by the Department of Defense (DoD) under award FA8750-17-C-0016. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the DoD.

References

[1] A. D. Sawadogo, T. F. Bissyand'e, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon, "Learning to catch security patches," *ArXiv*, vol. abs/2001.09148, 2020.

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *ACM Conference on Computer and Communications Security*, 2017, pp. 2169–2185.

[3] H. Ogawa, E. Takimoto, K. Mouri, and S. Saito, "User-side updating of third-party libraries for android applications," in *International Symposium on Computing and Networking Workshops*, 2018, pp. 452–458.

[4] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras, "From patching delays to infection symptoms: Using risk profiles for an early discovery of vulnerabilities exploited in the wild," in *USENIX Security Symposium*, Aug. 2018.

[5] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," *International Conference on Software Engineering*, pp. 771–781, 2012.

[6] S. Bratus. (2019) Broad agency announcement: Assured micropatching (AMP). https://govtribe.com/opportunity/federal-contract-opportunity/assured-micropatching-amp-hr001119s0089.

[7] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Network and Distributed Systems Security*, 2019.

[8] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna, "Spider: Enabling fast patch propagation in related software repositories," in *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, May 2020, pp. 531–548.

[9] A. P. Moore and A. D. Householder, "Multi-method modeling and analysis of the cybersecurity vulnerability management ecosystem," 2019.

[10] (1999) Common vulnerabilities and exposures (cve). MITRE. [Online]. Available: https://cve.mitre.org

[11] NIST, "National vulnerability database," https://nvd.nist.gov, NIST, 2000.

[12] A. Dulaunoy and P.-J. Moreels. (2012) cve-search - a tool to perform local searches for known vulnerabilities. http://cve-search.github.io/cve-search.

[13] (2020) Compare repositories - open hub. Synopsys. [Online]. Available: https://www.openhub.net/repositories/compare

[14] (2016) Version control systems popularity in 2016. RhodeCode. [Online]. Available: https://rhodecode.com/insights/version-control-systems-2016

[15] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *ACM Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[16] (2000) Vulnerability metrics. NIST. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[17] (2020, 02) Report a security or privacy vulnerability. [Online]. Available: https://support.apple.com/en-us/HT201220

[18] L. Torvalds. (2020) Linux. [Online]. Available: https://github.com/torvalds/linux

[19] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 345–355.

[20] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *USENIX Security Symposium*, 2019, pp. 869–885.

[21] T. Preston-Werner. (2013) Semantic versioning 2.0.0. [Online]. Available: https://semver.org/

[22] A. Arora, C. Forman, A. Nandkumar, and R. Telang, "Competition and patching of security vulnerabilities: An empirical analysis," *Inf. Econ. Policy*, vol. 22, pp. 164–177, 2010.

[23] A.-M. Jo, "The effect of competition intensity on software security an empirical analysis of security patch release on the web browser market," 2017.

[24] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," *IEEE Symposium on Security and Privacy*, pp. 374–391, 2018.

[25] T. Walshe and A. Simpson, "An empirical study of bug bounty programs," in *International Workshop on Intelligent Bug Fixing*, 2020, pp. 35–44.

[26] L. Ablon and M. C. Libicki, "Hackers' bazaar: The markets for cybercrime tools and stolen data," *Defense Counsel Journal*, vol. 82, pp. 143–152, 2015.

[27] L. Allodi, "Economic factors of vulnerability trade and exploitation," *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[28] A. Arora, A. Nandkumar, and R. Telang, "Does information security attack frequency increase with vulnerability disclosure? an empirical analysis," *Information Systems Frontiers*, vol. 8, no. 5, p. 350–362, Dec. 2006.

[29] A. M. M. Algarni and Y. K. Malaiya, "Most successful vulnerability discoverers: Motivation and methods," in *Conference on Security and Management*, 2013.

[30] F. Bisogni, "Proving limits of state data breach notification laws: Is a federal law the most adequate solution?" *Journal of Information Policy*, vol. 6, p. 154, 2016.

[31] J. Buckman, M. J. Hashim, T. Woutersen, and J. Bockstedt, "Fool me twice? data breach reductions through stricter sanctions," *Types of Offending eJournal*, 2019.

[32] S. Laube and R. Böhme, "The economics of mandatory security breach reporting to authorities," *Journal of Cybersecurity*, vol. 2, pp. 29–41, 2015.

[33] H. Cavusoglu, B. K. Mishra, and S. Raghunathan, "The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers," *International Journal of Electronic Commerce*, vol. 9, pp. 104 – 70, 2004.

[34] K. Campbell, L. A. Gordon, M. P. Loeb, and L. Zhou, "The economic cost of publicly announced information security breaches: Empirical evidence from the stock market," *J. Comput. Secur.*, vol. 11, pp. 431–448, 2003.

[35] L. A. Gordon, M. P. Loeb, and L. Zhou, "The impact of information security breaches: Has there been a downward shift in costs?" *Journal of Computer Security*, vol. 19, pp. 33–56, 2011.

[36] K. Vaniea and Y. Rashidi, "Tales of software updates: The process of updating software," *ACM Conference on Human Factors in Computing Systems*, 2016.

[37] A. Mathur, N. Malkin, M. Harbach, E. Péer, and S. Egelman, "Quantifying users' beliefs about software updates," *ArXiv*, vol. abs/1805.04594, 2018.

[38] P. Rajivan, E. Aharonov-Majar, and C. Gonzalez, "Update now or later? effects of experience, cost, and risk preference on update decisions," *Journal of Cybersecurity*, vol. 6, no. 1, 03 2020.

[39] O. Bergman and S. Whittaker, "The cognitive costs of upgrades," *Interactive Computing*, vol. 30, pp. 46–52, 2018.

[40] E. M. Redmiles, M. L. Mazurek, and J. P. Dickerson, "Dancing pigs or externalities?: Measuring the rationality of security decisions," *ACM Conference on Economics and Computation*, 2018.

[41] A. Mathur, J. Engel, S. Sobti, V. Chang, and M. Chetty, ""they keep coming back like zombies": Improving software updating interfaces," in *USENIX Symposium on Usable Privacy and Security*, Jun. 2016, pp. 43–58.

[42] K. Vaniea, E. J. Rader, and R. Wash, "Betrayed by updates: how negative experiences affect future security," in *ACM Conference on Human Factors in Computing Systems*, 2014.

[43] A. Sarabi, Z. Zhu, C. Xiao, M. Liu, and T. Dumitras, "Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state," in *PAM*, 2017.

[44] E. Rescorla, "Security holes . . . who cares?" in *USENIX Security Symposium*, 2003.

[45] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," *IEEE Symposium on Security and Privacy*, pp. 692–708, 2015.

[46] F. Li, L. Rogers, A. Mathur, N. Malkin, and M. Chetty, "Keepers of the machines: Examining how system administrators manage software updates for multiple machines," in *USENIX Symposium on Usable Privacy & Security*, 2019.

[47] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities," *ArXiv*, vol. abs/1905.09352, 2019.

[48] A. Ozment and S. E. Schechter, "Milk or wine: Does software security improve with age?" in *USENIX Security Symposium*, 2006.

[49] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *Conference of the center for advanced studies on collaborative research: meeting of minds*, 2008, pp. 304–318.

[50] S. Rastkar and G. C. Murphy, "Why did this code change?" in *International Conference on Software Engineering*, 2013, pp. 1193–1196.

[51] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *European Conference on Foundations of Software Engineering*, 2011, pp. 15–25.

[52] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on cvs," in *Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.

[53] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *International Conference on Computer Aided Verification*, 2012, pp. 712–717.

[54] G. Bavota, "Mining unstructured data in software repositories: Current and future trends," in *International Conference on Software Analysis, Evolution, and Reengineering*, vol. 5, 2016, pp. 1–12.

[55] E. A. Santos and A. Hindle, "Judging a commit by its cover," in *Workshop on Mining Software Repositories*, vol. 16, 2016, pp. 504–507.

[56] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," *IEEE International Conference on Software Maintenance and Evolution*, pp. 271–280, 2014.

[57] D. Binkley, "Using semantic differencing to reduce the cost of regression testing," in *Conference on Software Maintenance*, vol. 92, 1992, pp. 41–50.

[58] D. Binkley, R. Capellini, L. R. Raszewski, and C. Smith, "An implementation of and experiment with semantic differencing," in *Conference on Software Maintenance*, 2001, pp. 82–91.

[59] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, "Differential static analysis: opportunities, applications, and challenges," in *Workshop on Future of Software Engineering Research*, 2010, pp. 201–204.

[60] P. D. Marinescu and C. Cadar, "Katch: high-coverage testing of software patches," in *Joint Meeting on Foundations of Software Engineering*, 2013, pp. 235–245.

[61] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Conference on Automated software engineering*, 2010, pp. 33–42.

[62] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *Symposium on Foundations of software engineering*, 2008, pp. 226–237.

[63] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering*, 2011, pp. 1066–1071.

[64] M. Yang, J. Wu, S. Ji, T. Luo, and Y. Wu, "Pre-patch: Find hidden threats in open software based on machine learning method," in *SERVICES*, 2018.

[65] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, "The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques," in *Conference on Data Science in Cyberspace*, 2018, pp. 53–60.

[66] M. Sun, W. Wang, H. Feng, H. Sun, and Y. Zhang, "Identify vulnerability fix commits automatically using hierarchical attention network," *Endorsed Transactions on Security and Safety*, 5 2020.

[67] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss," in *International Conference on Dependable Systems and Networks*, 2019, pp. 485–492.