IEEE Security and Privacy Magazine - Dept. Building Security In
Editors: Fabio Massacci, John Steven

Technical Leverage: dependencies mixed blessing

Fabio Massacci
University of Trento and Vrije Universiteit Amsterdam

Ivan Pashchenko
University of Trento

***Abstract*—If modern software is a jungle of dependencies, how do we measure opportunities and risks of building your own software on somebody else software assets? Borrowing a concept from the 2008 financial crisis, we propose *technical leverage* as a simple yet effective metric.**

ONE YEAR AGO, Holger and Schroer captured in this column [1] the current crisis faced by software security quality control:

> The classic application software stack relied, at all levels, [...] on homegrown software developed by (often large) companies that had their own developers. [...] In the modern stack, coding [...] is increasingly reduced to only a small set of "glue code" or configurations built to combine software developed elsewhere, especially in the open source community.

They presented two neat software stacks side by side: a homegrown stack where everything is developed by a company and subject to several quality gates, and the modern stack where they pointed to the components that are now likely to be downloaded or integrated from the internet.

While a diagram of a software stack (Application, WebServer, DB, OS etc.) is 'technically correct', it gives a misleading impression of an orderly construction. The SolarWinds hack, recently discussed in this magazine, shows that Figure 1 is a better description of modern reality [2], [3].

If our state of practice is the one pictured there, how can we really measure our exposure? The quest for security metrics has a long tradition [10] but none really got traction.

So, we want to propose a metric that is *not* about security but it is about dependencies, by adapting a concept from the 2008 financial crisis: the more money you borrow from others, the more you can leverage and multiply your (limited) investment. Yet, if your lenders call back their cash, then you are bankrupt in a fortnight, as it happened to Lehman Brothers when their exposure with a 30-to-1 leverage turned sour [8].

Learning from Finance?

Allman's *Technical debt* [6] is a famous concept borrowed from finance: the consequence of a developer's action (more precisely inaction to fix) that may later require more maintenance.

 Published by the IEEE Computer Society

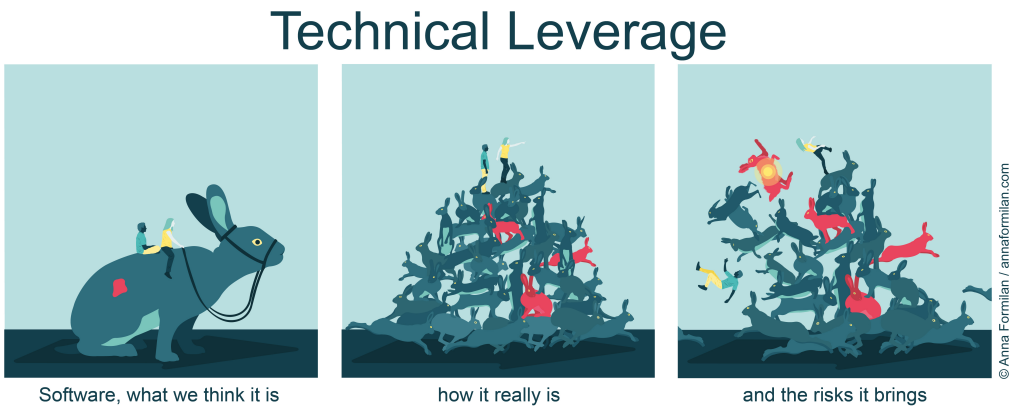


Figure 1. Technical leverage and the true nature of modern software

Table 1. Financial leverage vs Technical leverage

Financial leverage	Technical leverage
A company finances a new project with the help of borrowed money (debt)	Software developers reuse existing functionality from dependencies to focus only on new features in their projects
Financial leverage decreases corporate income tax liability and increases after-tax operating earning [4]	Using dependencies reduce time (i.e. cost) to develop new projects [5] and may increase performance (e.g. *numpy* in Python)
Debts imply interest rates [4] and must be eventually paid *or refinanced*, an observation absent in [6]	Dependencies must be monitored and updated (similarly to refinancing one's monetary debt) to avoid security vulnerabilities [7]
Financial leverage multiplies losses as well, which might lead to a crisis [8]	If managing dependencies becomes too costly, developers might stop updating them, implicitly accepting the risk of being exploited [9]

Yet, technical debt is inadequate to capture problems with dependencies because poorly written *own* code is the source of technical debts in a project [11]. Dagstuhl Seminar 16162, the closest proxy for an 'academic concept standardization meeting', scoped technical debt to the "internal system qualities" of a software project [12].

Let us review the arguments for the need of a new concept, all stemming from the same fact: *dependency code belongs to third-party projects*.

At first, *developers do not directly fix bugs in dependencies*. In theory, they could, and they could even join the other company or FOSS project and help them fixing the code. Along the same theory, your neighbor can help you washing the dishes and, for the most complicated bugs, can pay the college debts of your children. In practice, developers wait for a new version of the dependency fixing the bug (or the vulnerability) and update the dependencies of their project to this version. If the problem is in a transitive dependency, they have to wait while all intermediate dependencies are updated. In some cases, some newer, fixed transitive dependencies cannot be adopted as a direct dependency in the path is no longer maintained [13].

A major concern of developers is that *dependency updates could introduce breaking changes* to the dependent projects [9]. Method signatures could be changed and methods could be deleted. Empirical analysis [14] has show that the chances of a breaking update are 40-50%, a coin's toss.

Using software dependencies is an *opportunity*: one does not have to develop code, but simply use the good code that is there. Resources can be used elsewhere. Vulnerable dependencies can also generate *risks*, like the Equifax breach[1] that affected more than 147M people.

Technical Leverage

Here we just illustrated the key idea to spur the discussion in the community and refer to our conference paper [15] for technical details. Interested practitioners can also find an online demo for computing the proposed metrics for real-world FOSS software libraries: https://techleverage.eu/

We introduce the notion of **technical leverage** to assess the dependence on third-party functionalities. In finance it is the ratio between debt (other people's money in various form) and equity (one's own money). Similarly, in software it is

[1] https://epic.org/privacy/data-breach/equifax/

the ratio λ between other's people code (direct dependencies ℓ_{dir}, transitive dependencies ℓ_{trans}, possibly the baseline of programming language libraries ℓ_{std}) and one's own code ℓ_{own}.

$$\lambda = \frac{\ell_{dir} + \ell_{trans} + \ell_{std}}{\ell_{own}} \quad (1)$$

If one compares programs in the same ecosystem, the programming language/platform (e.g. Java and Maven) is the same across libraries so $\ell_{std} = const$. To compare libraries across different ecosystems (e.g. Python vs C libraries) the difference can be significant. Further, some libraries can be more mature than others and splitting leverage by type might be needed. Also in finance one distinguishes between different type of debts for a finer analysis. Table 1 compares financial and technical leverage.

Technical Leverage in Java/Maven

To understand the level of technical leverage in a FOSS ecosystem, we have measured it in some industry-relevant Java FOSS libraries corresponding to 10905 library instances (e.g. including widely used libraries such as org.slf4j:slf4j-api and org.apache.httpcomponents:httpclient).

Figure 2 we show only the technical leverage on *direct dependencies*, as the most critical metric, since developers typically react to the issues connected with their own code or their direct dependencies [9]. Transitive dependencies are also known to introduce security vulnerabilities [7] but are too complex to discuss them here.

Small-medium libraries with a code base less than 100K lines of code hare heavily leveraged: most library instances use a large code base of direct dependencies that is up to 10 000 times larger than their own size and 50% of small-medium libraries rely on 14.7 times bigger code base of their direct dependencies. We are not far from the money making 30-to-1 business of Lehman Brothers.

From the selected library sample, 50% of big libraries have direct technical leverage less than half (0.48) of their own size. So big libraries mostly come with their own code base.

From a developer perspective, we can show that *technical leverage actually pays off*: more leverage only add a modest delay of four days to the interval between releases. This is a good price to pay for shipping an application with almost 15x the code that you developed [15].

The interesting question from a security perspective is which are the risks? The answer is the same: we are not much farther from the abyss than Lehman Brothers

From the perspective of a user of a leveraged library, we can show that *technical leverage brings way more risks*: a more than average leveraged library has +60% chances of eventually including one (or usually more) vulnerabilities [15].

Figure 3 shows the number of vulnerabilities present in a library on the X axis and on the Y axis the distribution of the leverage of libraries having that number of vulnerabilities. A pattern is clear: with the exception of the very large number of vulnerabilities, corresponding to vulnerabilities present in very large libraries with over 100KLoC, as leverage increase so does the number of vulnerabilities.

What's next?

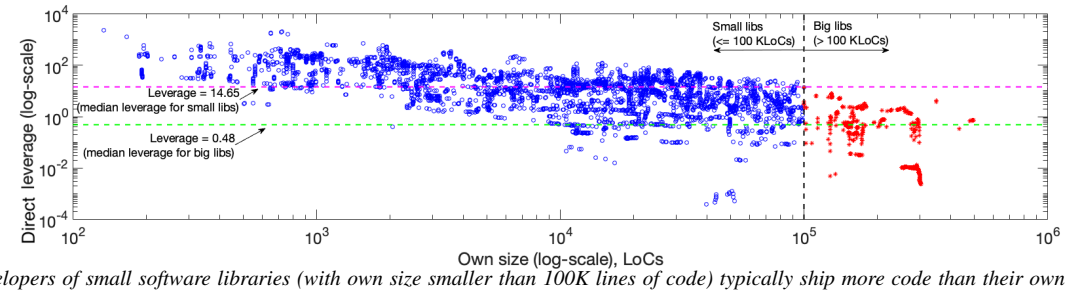
Technical leverage is a relatively simple metrics to capture the dependence of a software library on third-party code. It shows we are very close to the leverage of Lehman Brothers: we mostly ship code that is not ours. This is both a great opportunity and a huge risk.

Indeed, this metric shows that *we have a major problem as a community, a classical moral hazard*: those reaping the benefit of technical leverage (a library's developers) are not the same people exposed to the corresponding risk (the users of the library). A different regulatory regime for software liability might be needed.

Additional details and security metrics to measure the nature and directions of changes when a library is updated are in the conference paper [15] and on the website: https://techleverage.eu/. Let us know what you think.

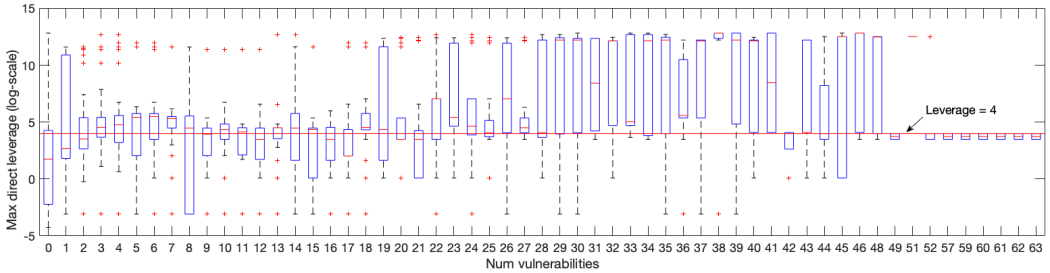
Acknowledgement

This work was supported in part by the European Commission by grants num. 830929 (H2020-CyberSec4Europe – https://cybersec4europe.eu) and num. 952647 (H2020-AssureMOSS – https://assuremoss.eu). Artwork in Figure 1 by Anna Formilan - https://annaformilan.com.



Developers of small software libraries (with own size smaller than 100K lines of code) typically ship more code than their own direct leverage > 1). For the majority, their own code is only a small fraction of the overall codebase (less than 6%, corresponding to a median direct leverage of 15). In other words, they ship mostly somebody else code. The direct leverage of large libraries ($> 100K$ lines of code) is much smaller than the size of their own code and hardly exceed 2, i.e. there is at least 33% of own code.

Figure 2. The direct leverage in comparison to the own size of a library



Direct leverage equal to 4 allows visual separation between the libraries exposed to high number of vulnerabilities vs libraries exposed to a small number of security vulnerabilities in our library sample. The only exceptions are the handful of libraries at the extreme right. They are libraries with large own code base $\geq 100K$ LoCs which are always affected by security vulnerabilities (mostly in their own code base) just because of their size.

Figure 3. Max direct leverage per library vs Number of vulnerabilities in a library version

REFERENCES

1. H. Mack and T. Schroer, "Security midlife crisis: Building security in a new world," *IEEE Security & Privacy*, vol. 18, no. 04, pp. 72–74, jul 2020.
2. M. Pittenger, "Open source security analysis: The state of open source security in commercial applications," Black Duck Software, Tech. Rep., 2016.
3. J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" Ph.D. dissertation, TU Delft, Computer Science, 2015.
4. A. Kraus and R. H. Litzenberger, "A state-preference model of optimal financial leverage," *J Finance*, vol. 28, no. 4, pp. 911–922, 1973.
5. M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio, "An industrial case study on reuse oriented development," in *Proc. of ICSME'05*. IEEE, 2005, pp. 283–292.
6. E. Allman, "Managing technical debt," *Commun. ACM*, vol. 55, no. 5, 2012.
7. R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Emp. Soft. Eng. Journ.*, May 2017.
8. K. Berman and J. Knight, "Lehman's three big mistakes," *Harvard Business Review*, September 2009.
9. I. Pashchenko, D. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proc. of CCS'20*. ACM, 2020.
10. A. Jaquith, *Security metrics: replacing fear, uncertainty, and doubt*. Pearson Education, 2007.
11. A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. and Softw. Tech. Journ.*, vol. 64, pp. 52–73, 2015.
12. P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
13. I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *TSE*, 2020.
14. J. Huang, N. Borges, S. Bugiel, and M. Backes, "Up-to-crash: Evaluating third-party library updatability on android," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 15–30.
15. F. Massacci and I. Pashchenko, "Technical leverage in a software ecosystem: Development opportunities and security risks," in *Proc. of ICSE'21*, 2021.

Fabio Massacci is currently with University of Trento, 38123 Trento, Italy and Vrije Universiteit of Amsterdam, 1081 HV Amsterdam, The Netherlands. MSc'93, PhD'98. He is interested in foundational and experimental approaches to security. In 2015 he received the 10 years Most Influential Paper Awards by IEEE Requirements Engineering Conference for his work on security in socio-technical systems. He is currently the coordinator of the H2020 AssureMOSS project on the security of multi-party open source software. He is member of the ACM, American Economic Association, IEEE, and Society for Risk Analysis. Contact him at *fabio.massacci@ieee.org*.

Ivan Pashchenko is currently with University of Trento, Italy. MSc'12, PhD'19. He is interested in open-source software security, software verification, and machine learning for security. In 2017 he was awarded a Silver medal for the second place at the ACM/Microsoft Student Research competition in the graduate category. He is the UniTrento main contact in *"Continuous analysis and correction of secure code"* work package for the H2020 AssureMOSS project. He is member of the ACM. Contact him at *ivan.pashchenko@unitn.it*.