Uniform Parameter Scoring System

Yehuda Katz

Abstract

System administrators are usually trusted to be experts on the systems they control, yet security breaches and performance problems can often be traced to improper configuration by these same administrators. These configuration mistakes aren't made deliberately and certainly not maliciously, but are usually due to a lack of information about the consequences and interactions of these settings. We call this *Uninformed Configuration*. Existing research focuses on ensuring users don't make security mistakes or helping developers/engineers validate their systems meet security and performance targets, but there is little research focusing on the system administrator who bridges these two groups, taking systems from developers and maintaining them for the end user.

The Common Vulnerability Scoring System is used to assist administrators in understanding the actual risk of a security vulnerability and the impact the particular vulnerability can have on their systems. We present a scoring system that can be used to label settings for a program to explain the severity and potential impact of changing each setting.

Background

The Common Vulnerability Scoring System (CVSS) is the most well known standard for evaluating the severity of security vulnerabilities. CVSS has received a lot of criticism for its accuracy and effectiveness and other scoring systems exist, but performance of CVSS has been studied [12] and CVSS continues to be used as the main scoring system for vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database. The initial CVSS standard was created under the stewardship of the National Infrastructure Advisory Council, an advisory group under the United States Secretary of homeland Security. The initial CVSS version 1 was released in 2005 with no peer review, but has since been managed by the Forum of Incident Response and Security Teams and has undergone regular updates. CVSS version 4 was released in November 2023, but as of this writing has not been adopted/integrated into existing vulnerability reporting systems.

> **CVSS in its own words [1]**
>
> The Common Vulnerability Scoring System (CVSS) captures the principal technical characteristics of software, hardware and firmware

vulnerabilities. Its outputs include numerical scores indicating the severity of a vulnerability relative to other vulnerabilities.

System administrators are usually trusted to be experts on the systems they control, yet security breaches and performance problems can often be traced to improper configuration by these same administrators. These configuration mistakes aren't made deliberately and certainly not maliciously, but are usually due to a lack of information about the consequences and interactions of these settings. We call this **Uninformed Configuration**. Because many vulnerabilities, as well as performance problems, are caused by Uninformed Configuration, we explore what a standard might look like that could inform administrators about the effects of making settings changes. We model this standard on the CVSS version 3 because it will be familiar to system administrators and already defines many of the primitives we need for our Uniform Parameter Scoring System (UPSS, pronounced OOPS).

Comparison to other fields

A complex software package can have thousands of settings that will change how the program functions in ways ranging from trivial to extremely beneficial or catastrophic. After a complex software package, an area that might have the next highest number of "knobs and switches" - and an area where improper configuration certainly has a more-immediate impact on life and death - is a modern jet airplane cockpit [2]. Sherry, et al. [14] and Singer [15] describe a standard design language for a modern airplane and research into how to evaluate cockpit system usability. A lot of the design of modern cockpits is the result of hindsight after various disasters, but many disasters result in additional checklist items that pilots must do rather than changes to the design of the airplane. There does not appear to be a written standard across the industry for switch/knob design (although the European Aviation Safety Agency Certification Specification 25 addresses the issue [6]), but we can flag some things that are universal that will give us a greater understanding of evaluating risk [11]. For example, if a toggle switch can do two actions, the safer action is usually towards the bottom. If the action of a switch is not reversible or could cause significant negative impacts to flyability, the switch will be covered or require multiple hand motions to activate (e.g. turn and pull). Karpanos, et al. show that people evaluate the quality of a product or system based on the perceived quality of the control knobs [13]. We can't directly apply these switch design paradigms to software configuration because changing one setting in a text file is just as easy as changing any other setting - there isn't a simple way to put a virtual "cover" over a dangerous configuration option.

At the opposite extreme, alert fatigue is a well studied and understood condition in which receiving too many warnings, especially warnings that claim to be serious, can cause users to completely ignore all warnings. The same applies to having too many settings that can be changed [16] and certainly to attaching warnings to every single knob. Do you know anyone who looks at a California

Proposition 65 warning and takes it seriously? If you had to acknowledge that a coat hanger can be dangerous and should not be given to children every time you went to use a coat hanger, most people would probably dump their coats on the floor.

Labeling too many settings as critical for security or performance will reduce the overall usefulness of the entire system. Not labeling all the settings that really make a difference, or not explaining the potential outcomes properly, will mean this scoring system is not useful. The key is to be able to find the settings that really matter and to communicate their true impacts appropriately. We also can not create a framework to prevent people from purposely doing dangerous things. When you purchase a trampoline, it does not come with a warning that says "Do not install next to the opening of a running wood chipper". [8] The purpose of UPSS is to assist administrators in understanding the impacts of their choices and prevent uninformed choices, but not to prevent all security and performance problems.

Simplification and Demonstration of Need

Xu, et. al. [16] suggest guidelines for simplifying configuration options by eliminating unnecessary parameters and reducing the possible settings for the remaining parameters. Anecdotally, developers and administrators are resistant to the appearance of the removal of granularity as administrators believe that they are competent to control all the possible parameters. Xu, et. al. conclude that existing search options for understanding available configuration options are not sufficient, including noting that naive keyword search, Google search (or other search engines), and Natural Language Processing-based querying are not sufficient general solutions. A standardized scoring system for the importance of each parameter will improve the chances of an administrator finding the knobs that are actually important.

How Configuration Affects Performance and Security

It seems obvious that changing operational parameters can cause significant changes to the operation of a program, but we need a strong framework to evaluate the types of changes in order to draw conclusions about the changes to the settings themselves. We will create our initial framework using a generic web server as our subject.

In all of the documentation we consulted while writing this paper, configuration options are listed in alphabetical order which no attention drawn to the importance or consequences of changing specific settings. Some programs have a getting started document which may say something like "The primary server is fairly resource intensive, and must be installed on a robust, dedicated server." [7],

but no specific notes about performance, or might have a "Performance Tuning" document [3] buried in alphabetical order with the rest of the settings.

`"Effect Categories"` of Settings

1. Settings which only affect a Display value (someone says "Hey, that's weird.") (GUI)

This includes any value which is only displayed to a user, but does not change any part of the way the program runs. These settings are generally not critical in any way to the operation of the program, although they can have external effects. These will generally have the lowest score.

Example: Administrator's email address - The Apache HTTPD Server exposes the `ServerAdmin` configuration value as an environment variable for CGI applications and also prints the value on error pages. This value is not used anywhere else in the server. If an administrator includes an email address in this value, there is a risk that it will be harvested by spammers, but that is completely external to the operation of the program.

2. Settings which affect the output of the program, but not performance or security (the client says "Hey, it's broken!") (OUT)

This includes any value which materially changes the operation of the program from the perspective of the user, but does not actually change the code paths the program runs or the system resources being used.

Example: Wrong backend - The HAProxy load balancer allows the creation of multiple `frontend` servers, each pointing to one or more `backend` servers. HAProxy itself doesn't care which backend you connect to for each frontend, but choosing the wrong backend will present the web client with the wrong website. (Note: For this example, we are assuming that the backend applications handle their own security or are intended to be public, but in either case are not relying on the frontend proxy for security purposes.)

3. Settings which affect core functionality (the administrator says "Hey, it's broken!") (FNC)

These are settings which will make the program obviously broken if they are set wrong, but are not tied directly to major performance or security changes. These setting could cause *other* settings to be more important for security or performance, but don't make a difference by themselves.

Example: Socket Bindings - If a service is intended to be exposed to the network, it must be exposed to the correct network interface and port or it will not be accessible. Alternatively, if the program is intended for internal use and is bound to a TCP port instead of a Unix Domain Socket, it adds overhead of TCP connections with no benefits. This can affect performance in minor ways, for

example, network traffic overhead or kernel autobind limits (the modern Linux kernel has a limit of 2^{20} which is more than enough for most current use cases, but some systems can have a lower value and having a large number of entries can cause performance problems or denial of service when approaching the limit).

Example: Dynamically Loaded Extensions - Many programs load extensions through Dynamically Linked Libraries (Windows)/Shared Objects (Linux). Without these extensions, the server might not start at all, or might show errors (for example, if an unrecognized configuration directive is present) or unprocessed code (if no extension is loaded to process a particular file type) instead of proper web pages.

Deciding if a setting qualifies for this group While it again depends on the specifics of the program, you might think that loading extensions would affect security and performance because more code is potentially being executed. For purposes of this example, loading an extension in the web server to allow processing dynamic web pages (i.e. PHP) is a core part of the functionality required and the server can not be considered functioning without it, so the extension itself is assumed to be properly programmed and loading of the extension itself is considered to not affect performance. However once the extension is loaded, other settings become more important; for example, if PHP is loaded as an Apache HTTPD module, it adds settings which can change its behavior and affect performance or security, but loading the extension itself is safe and performant.

4a. Most critical (performance) (PRF)

These settings may directly affect the amount of system resources a program is allowed to use or apply limits to the amount of resources used by any particular portion of the program.

Example: Number of sockets, number of threads, memory limit, concurrency limit - HAProxy will open file descriptors for each connection in and out of the load balancer. By default, HAProxy limits the number of inbound connections to a small number to make sure it will work on all systems and it sets the kernel `ulimit` to prevent opening more sockets than it expects to use. It appears to an administrator that the service is being limited by a low socket limit and the administrator might try to raise the limit using standard service controls (i.e. SystemD Unit File); because the server is controlling that value, it won't resolve the issue. The same issue can happen with low defaults for maximum number of threads, especially as some programs are originally created without thread support and add threading "experimentally" as they are developed, so it is not enabled by default and the program doesn't take advantage of the large number of processor cores available in modern servers. The reverse is also true - a program with no memory limits or no concurrency limits can bring down an entire server running multiple services if a single service runs away with its resource usage.

4b. Most critical (security) (SEC)

These settings will directly affect the security of a system. These are arguably the hardest category to find (especially as we are excluding problems arising from improper coding and any setting that would break core functionality of the program).

Negative Example: Heartbleed - The Heartbleed security vulnerability was caused by OpenSSL not validating the length of data requested by a client, allowing it to return data from memory outside the expected location. Most software linked with vulnerable versions of OpenSSL did not provide a way to disable heartbeats as a workaround to the vulnerability, so the only "workaround" was to disable SSL completely. Because there was no setting an administrator could change that would not break the server, this issue would not fall into this category.

Negative Example: Hiding Version Numbers - Many vulnerability scanners report security vulnerabilities solely based on the version a program announces (for example, through a `Server` or `X-Powered-By` header). Internet bounty seekers and script-kiddy attacks also commonly use these values before checking for actual vulnerabilities as the check or exploit can be difficult or expensive. Hiding the version number of the software might mask these security vulnerabilities from low-skilled attackers, but does nothing to change the actual state of security of the program.

Possible Example: Symlinks - Apache HTTPD lets the administrator configure whether it will follow symlinks on the file system when serving data. On a multi-user system, symlinks can be abused to load protected files that a regular user is not allowed to read, but that HTTPD might be able to read if it is running as a privileged user. HTTPD has a workaround that makes this harder: the administrator can enable `SymLinksIfOwnerMatch` which will only allow the server to serve the file if the source and target of the link match. [4] This check is susceptible to a race condition if the destination of the symlink is changed between when the ownership is checked and when the file is opened for reading.
(Possible) Example: AllowOverride - Apache HTTPD lets the administrator configure whether it will read configuration files from all directories, and if so, which directives to allow. This allows an unprivileged user to fundamentally change the operation of the server.
All of the above settings could be flagged as impacting security by someone knowledgeable about the software, but might be hard to find using an automated tool.

Example: SSL/TLS/etc. Ciphers - Most programs allow the administrator to select which ciphers will be used for client connections. Many, if not most, administrators accept the default values unless they have problems making connections or run an auditing tool which flags outdated ciphers, but the default values shipped by application developers are often outdated or are just as uninformed. Software packages that ship from operating system maintainers

are often even further behind because they don't backport upstream changes regularly.

Assumptions

Determining the correctness of application code is outside the scope of the scoring system, therefore we will assume that a program is properly coded and changing a setting will not cause a security or performance issue due to improper coding. For example, it is possible that changing a display value could cause a buffer overflow or underrun and allow access to unintended parts of the memory. Concerns in this area vary widely, and could possibly not be applicable or be entirely resolved, depending on the language a program is written in (for example, a memory-safe and type-safe language).

How Knobs Can Be Found and Scored

The simplest way to score configuration options is to have the developer flag each parameter with a value as they develop the program. While effective at creating an initial scores, this method does nothing to ensure values are kept up to date - unlike security vulnerabilities, which do not regularly change once announced, the effects of configuration parameters can vary as a program continues to be developed. Existing programs can also have thousands of knobs that would need to be labelled. Automated methods to assist in labelling would potentially handle both of these concerns, but would need to be validated.

Static Analysis

Some characteristics lend themselves more obviously to static analysis than others.

The Scoring Standards

The Uniform Parameter Scoring System (UPSS) captures the effects of changes to software configuration parameters. It outputs a numerical score indicating the significance of the consequences of changing the parameter. The v1 score of a parameter is most useful when compared to the scores of other parameters of the same program.

We propose these metrics as a starting place and plan to adapt and revise as there is more real-world use, similar to the eventual development path taken by CVSS. The development of these standards was inspired by CVSS version 3.1 although there are significant changes to the way CVSS version 4.0 is calculated which are supposed to make it simpler. [10] (The new CVSS scoring method relies on expert opinions collected from previous scores and thus is not possible with a newly created system.)

Initial Metrics

Metrics can be broken down into two categories: the changes intrinsic to the setting itself, and "external" considerations for the administrator choosing the correct value.

Intrinsic Characteristics

1. Affects Processor Usage (PU) This metric measures the changes to the CPU load expected from changing a setting.

Impact	Description
Critical (C)	Changing this setting is expected to have an impact on CPU usage to the point of likely significantly (positively or negatively) impacting overall performance.
High (H)	Changing this setting is expected to have a measurable impact on CPU usage to the point of potentially significantly (positively or negatively) impacting overall performance.
Low (L)	Changing this setting is expected to produce a noticeable, but not critical, change in performance.
None (N)	Changing this setting is not expected to produce any (measurable) changes to performance.

Scoring Guidance: `Affects Processor Usage` should include settings which change the expected processor cycles as well as the thread or process count.

Points of interest for Static Analysis:

- A `Low` score might be appropriate if a setting changes a significant number of lines of code executed.
- A `High` score might be appropriate if a setting directly changes the number of processes, threads, or arbitrary usage controls (i.e. `nice` parameters)
- A `Critical` score might only be able to be appropriately decided by an experienced developer or administrator.

2. Affects Memory Usage (MU) This metric measures the changes to the RAM/SWAP usage expected from changing a setting.

Impact	Description
Critical (C)	Changing this setting will allow unrestricted memory usage.
High (H)	Changing this setting is expected to cause a calculated, "human" measurable (i.e. `free -h`, gigabyte level) change in memory usage.
Low (L)	Changing this setting is expected to cause a change in memory usage, but not at a scale significant enough to notice.

Impact	Description
None (N)	Changing this setting is not expected to produce any (measurable) changes to memory usage.

Scoring Guidance: `Affects Memory Usage` is `Critical` if a setting can be changed that would allow run-away memory usage to the point of crashing the entire system and other reasonable values could be set which would prevent such a situation. If there is no value for the setting to limit memory usage, the setting is not automatically critical.

3. Affects Socket Usage (SU) This metric measures how the program opens sockets/file descriptors, how many it can open, and in what locations.

Impact	Description
Yes (Y)	Changing this setting will change the number of open files, types of open sockets, or locations of open files.
No (N)	Changing this setting will not affect file/socket handling.

Points of interest for Static Analysis:

- Network sockets
- Any file handling operations
- Calls to change `ulimit`

4. Affects Input Sanitizing (IS) This metric measures changes to how the program sanitizes user input.

Impact	Description
Yes (Y)	Changing this setting will change whether user input is trusted and/or how it is sanitized for dangerous values.
No (N)	Changing this setting will not affect input handling.

5. Does changing this setting cause the scores of other settings to change? (CO) This metric measures whether this setting doesn't fall into any of the above categories itself, but changing it will cause changes to the other metrics.

Impact	Description
Yes (Y)	Changing this setting will likely directly change other metrics.
No (N)	Changing this setting will not directly affect other metrics.

Scoring Guidance: A good example of this is the Apache HTTPD MPM model and `.htaccess` processing. Besides that, this is open-ended.

Points of interest for Static Analysis:

- Loading DLLs/extensions

External Characteristics

1. Difficulty of Determining the Correct Value (CV) This metric measures how hard it will be for an administrator who has never used this software to choose the correct value for this setting.

Impact	Description
Critical (C)	An administrator experienced in other areas is still likely to have difficulty choosing the correct value for this setting.
High (H)	An administrator experienced in other areas is likely to choose the correct value for this setting, but a less experienced administrator will have difficulty.
Low (L)	Any administrator will be able to choose the correct value with minimal difficulty.
None (N)	The value of the setting has no practical changes to the program operation.

Scoring Guidance: Settings in the `Display Only` category are likely to fall into the `None` value of the metric. Settings with a fixed range of values which are easy to test and change (see the next metric) may also have a `None` value.

Points of interest for Static Analysis:

- Consider other possible connections (or lack of connections) between difficulty of setting a value and its `effect category`.

2. Difficulty of Changing the Value Later (CL) This metric measures the difficulty of changing a setting once it has been set and the program has run.

Impact	Description
Critical (C)	Once this setting has been set, it may not be possible to change it.
High (H)	Once this setting has been set, it will be a complicated or lengthy process to change it, or it may require downtime to change.
Low (L)	Once this setting has been set, the complexity of changing it may vary, for example, depending on the length of the the program has been run (i.e. built-up data)
None (N)	This setting can be changed easily and at any time.

Scoring Guidance: Consider whether it will be difficult to change this setting later, either because the steps required to change it are complex or not documented, or because it will require significant effort, or because there is a high likelihood of the change going wrong and causing damage to the system or data because it will directly invalidate or render unreadable previously stored data.
This could also apply to software licensing tied to a particular hardware/software "thumbprint" (i.e. Windows activation or Proxmox activation) where it will not be possible to make changes without invalidating the license (this would likely need to be flagged by a developer, although static analysis of the licensing code could also find critical values).

Points of interest for Static Analysis:

- Encryption settings - data encrypted at rest may be difficult or impossible to convert to a new encryption method.

3. Frequency of Change (FQ) This metric measures whether a setting is often changed. (Obscure settings are often more dangerous and harder to get support for.)

Impact	Description
Never (N)	This setting is only used for specific uses (i.e. debugging during development) and is not intended for regular use.
Low (L)	This setting is usually only changed by experts on this program.
High (H)	This setting is commonly changed, but may require expertise to choose the correct value.
Always (A)	Anyone who uses this program is expected to change this setting.

Scoring Guidance: This is like the box on your tax form which says "this will not apply to most people".

Points of interest for Static Analysis:

- Is it possible to find a major deviation from a standardized configuration

Communicating the Results

Qualitative Severity Rating Scale

It can be useful to map a numeric score to a textual representation. For the purposes of this initial version, we will adopt the scale and use policy used by CVSS. [5]

UPSS Score	Rating
0.0	None
0.1 - 3.9	Low
4.0 - 6.9	Medium
7.0 - 8.9	High
9.0 - 10.0	Critical

As an example, an UPSS Base Score of 5.0 has an associated severity rating of Medium. The use of these qualitative severity ratings is optional and there is no requirement to include them when publishing UPSS scores. They are intended to help administrators properly assess the consequences of a value change.

Vector String

As inspired by CVSS, the values of each metric can be quickly communicated/transferred with a Vector String.

The UPSS v0.9 vector string begins with the label "UPSS:" and a numeric representation of the current version, "0.9". This is followed by `/EC:` and the abbreviation of the `Effect Category` of the setting. Metric information follows in the form of a set of metrics, each preceded by a forward slash, "/", acting as a delimiter. Each metric is a metric name in abbreviated form, a colon, ":", and its associated metric value in abbreviated form. The abbreviated forms are defined above (in parentheses after each metric name and metric value).

For example, a setting which affects only a displayed value, has no effect concerns, is easy to change, and is commonly changed would produce the vector: `UPSS:0.9/EC:GUI/PU:N/MU:N/SU:N/IS:N/CO:N/CV:N/CL:N/FQ:A`.

A setting which turns on debug logging to disk with multiple separate files concurrently which will cause a significant impact to performance might produce the vector: `UPSS:0.9/EC:PRF/PU:C/MU:L/SU:Y/IS:N/CO:N/CV:N/CL:N/FQ:N`

Metric Values and Scoring Equations

The formula for calculating the final numerical score is created by taking multiple examples of settings from programs we are very familiar with, deciding what qualitative rating we would expect them to receive, and assigning weights to each factor so as to achieve the expected results. The scoring algorithm for UPSS version 1 is still being developed. An excel spreadsheet is available with more calculation details. [9]

CVSS v3.1 assigned values to each of their metrics using metrics based on real world vulnerability and usage data which is not (yet) available for UPSS. CVSS v4.0 makes the math even more complicated using multiple lookup tables which are also produced with the hindsight of almost 20 years of data.

The total score is built from three components:

1. `Base-Score`: The metrics related to the effects of the setting itself provide a base score This is currently all of the `Intrinsic` metrics
2. `Effect-Score`: The `Effect Category` of the setting provides a modifier based on the type of impact of the setting
3. `Change-Score`: The metrics related to how difficult the setting is to correct/change This is currently all of the `External` metrics

The scoring method will be inserted here for UPSS version 1.

Future Work / Expected Revisions

Analysis of Interactions

It is out of scope of this work to determine a standard for evaluating how changes to one parameter dynamically affect the score of another parameter. We leave this to future work.

Making Recommendations for Proper/Optimal Values

Unlike vulnerability disclosure, we can also make suggestions to the administrator of preemptive measures that can be taken based on the known scores of multiple service running on a single computer. We leave this to future work.

Bibliography

[1] 2023. *Common Vulnerability Scoring System version 4.0: Specification Document*. Retrieved December 13, 2023 from https://www.first.org/cvss/v4.0/specification-document

[2] 2023. Interview with Boeing Product Manager.

[3] Apache Performance Tuning. Retrieved from https://httpd.apache.org/docs/2.4/misc/perf-tuning.html

[4] cPanel - HTTPD Symlink Race Condition Protection. Retrieved from https://docs.cpanel.net/ea4/apache/symlink-race-condition-protection/

[5] CVSS v3.1 Specification Document - Qualitative Scale. Retrieved from https://www.first.org/cvss/v3.1/specification-document#Qualitative-Severity-Rating-Scale

[6] *European Aviation Safety Agency Certification Specifications for Large Aeroplanes CS-25*. Retrieved from https://www.easa.europa.eu/en/document-library/certification-specifications/group/cs-25-large-aeroplanes#cs-25-large-aeroplanes

[7] Puppet 7 Server - System requirements. Retrieved from https://www.puppet.com/docs/puppet/7/system_requirements#system_requirements

[8] *Trampoline Surronded By Broken Glass (AI)*. Retrieved from https://www.cs.umd.edu/~yakatz/research/upss/a-boy-jumping-on-a-trampoline-surrounded-by-broken-glass.png

[9] UPSS Calculator v0.9. Retrieved from https://www.cs.umd.edu/~yakatz/research/upss/calc-v0.9.xlsx

[10] Dave Dugal and Dale Rich. Announcing CVSS v4.0. In *35th Annual FIRST Conference*. Retrieved December 31, 2023 from https://www.first.org/cvss/v4-0/cvss-v40-presentation.pdf

[11] David Jensen. Product Focus: Cockpit Switches. *Aviation Today*. Retrieved from https://www.aviationtoday.com/2004/03/01/product-focus-cockpit-switches/

[12] Pontus Johnson, Robert Lagerstrom, Mathias Ekstedt, and Ulrik Franke. 2018. Can the Common Vulnerability Scoring System be Trusted? A Bayesian Analysis. *IEEE Trans. Dependable and Secure Comput.* 15, 6 (November 2018), 1002–1015. https://doi.org/10.1109/TDSC.2016.2644614

[13] Evangelos Karapanos, Stephan Wensveen, Bart Friederichs, and Jean-Bernard Martens. 2008. Do knobs have character?: Exploring diversity in users' inferences. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, April 05, 2008. ACM, Florence Italy, 2907–2912. https://doi.org/10.1145/1358628.1358782

[14] Lance Sherry, Peter Polson, and Michael Feary. 2002. Designing User-Interfaces for the Cockpit: Five Common Design Errors and How to Avoid Them. November 05, 2002. 2002-01-2968. https://doi.org/10.4271/2002-01-2968

[15] Gideon Singer. 2002. Methods for Validatng Cockpit Design The best tool for the task. PhD thesis. Institutionen för flygteknik. Retrieved from https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A9102&dswid=-8939

[16] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, August 30, 2015. ACM, Bergamo Italy, 307–319. https://doi.org/10.1145/2786805.2786852