From Aviation Safety to Cyber Resilience

Leveraging aerospace industry’s century-long safety culture for cybersecurity

Summary
Software development has a poor safety record. The number of cyberattacks hat risen dramatically since 2004, clearly outrunning the growth of the internet itself. In parallel, aviation has a proven track record of achieving the highest level of safety and serves as an example even in the information technology industry itself. Despite increasing software complexity, the existing certification processes for airborne equipment manage to assure extremely reliable software. The more recent certification process for information technology products applies similar methods of formal development but is not as widespread as its aviation counterpart. More importantly, the aerospace safety culture is missing. As such, information technology can benefit from applying the aerospace industry’s methods, procedures and mindset to its development process. Due to the similarities of both formal development worlds, the transfer will be easier than it may seem.

History repeating I

It started Tuesday June 27th, 2017. In the offices of A.P. Møller-Maersk around the world, computer screens started to go black. Within a few hours, global operations of the world’s largest container shipping company — accounting for 20% of the market — had been stopped. It took the company ten days to revive its computer system; ten days, during which operations were mainly done by hand. And had it not been for a power outage in Ghana, causing a single computer to have been offline when disaster struck, thus retaining a sole copy of crucial infrastructure data, it would have been even worse (see Greenberg, 2018, for the full story on Maersk).

Incidents are almost always the result of an unfortunate concatenation of weak spots, human error and negligence, and design flaws. Plus a bit of bad luck. In Maersk’s case, the latter was the use of a single computer with accounting software in the Ukraine — nothing that could reasonably have been prevented, unless you are a fundamentalist system administrator who forbids any software that is not rolled out through the central office (if you are, make sure you are not accountable for operational success and profit). But the design flaw was of a different order of magnitude. Dubbed EternalBlue, it was a highly critical vulnerability in Microsoft Windows, allowing ransomware like NotPetya —

the one that hit Maersk — to spread throughout a network without human interaction. But no-one could have known. Or could they?

It started Friday May 12th, 2017, early in the morning. By the end of the day, more than 200,000 computers all around the world had been affected. The banking industry of Spain seemed to have been a hot spot, likewise the national health system in the UK. German railways were affected, too, displaying red ransomware messages on passenger information screens at stations across the country. And had it not been for a single hacker, who found a kill switch in the form of an unregistered internet domain, the damage caused by WannaCry would have been even larger.

Let’s cut it short. No-one could have known. Or could they?

The vulnerability had been publicly announced in April 2017; a patch by Microsoft had been available even before that, albeit not for Windows XP, for which security support had ceased. Which takes us another step back: human error and negligence. The thing that could reasonable have been prevented was continuing to use outdated, unsupported and unpatched Windows XP machines beyond their expiry dates.

History repeating II

But there is no safety culture in software. Software is cheap. Software needs to be updated anyway. Software vendors are not liable for anything. Software cannot do any physical harm. As long as we do not build our critical infrastructure on software, as long as our economy does not depend on it, as long as we do not network security systems or medical equipment — what can go wrong? Oh, wait…

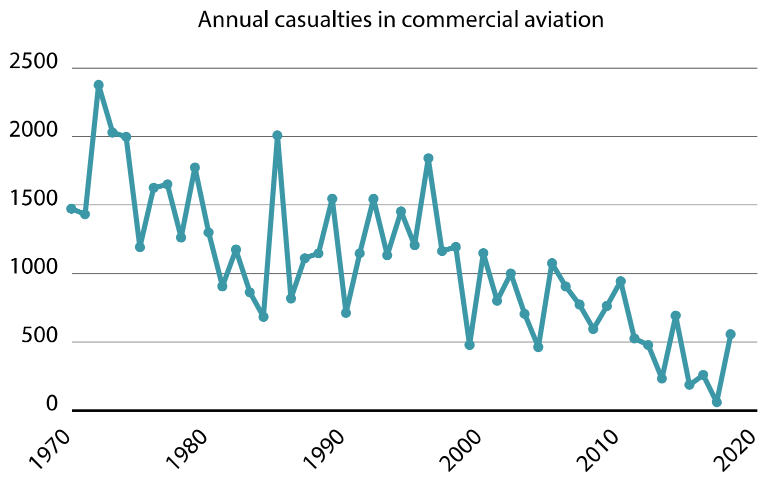
Weak spots, human error and negligence, design flaws and bad luck hit aviation, too. On March 10th, 2019, Ethiopian Airlines flight 302 crashed six minutes after take-off, killing all 157 people aboard. On October 29th, 2018, Lion Air flight 610 had crashed 12 minutes after take-off, killing all 189 people aboard. Both flights were operated by a Boeing 737 MAX series aircraft, which had entered into service on May 22nd, 2017.

In hindsight, one may question the response after the first accident. Yet, the Lion Air crash was the first indication of a fundamental problem with the design of the new aircraft and thorough investigations were started, leading to a first report in November 2017 and improvement work at Boeing’s. The similarities between the Ethiopian Airlines and the Lion Air accidents were immediately evident when the second happened. Many operators stopped operating their 737 MAX aircraft the day after, and the global fleet was grounded within a week. The aircraft will not fly again until the design flaw has been removed. The airframe manufacturer is responsible for offering the solution and each individual aircraft will not take off before the necessary patches have been applied.

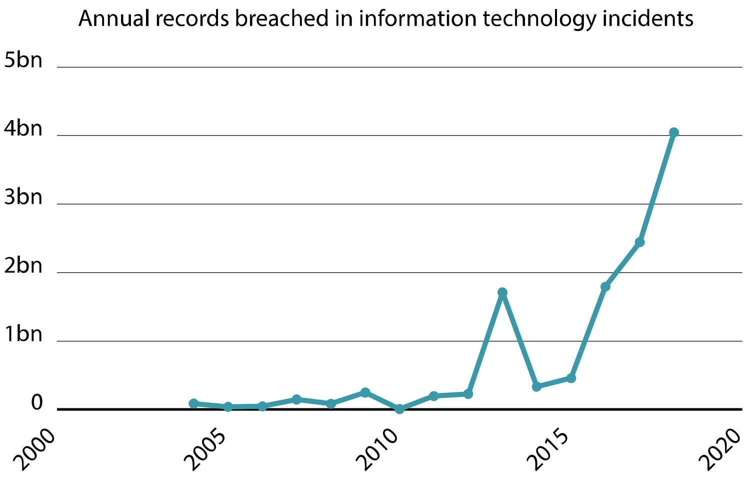
Safety culture

There *is* a safety culture in aviation — although one is tempted to say: there *still* is, as the 737 MAX story represents a narrative of fundamental flight mechanical changes to an existing, originally extremely mature and safe airframe, the consequences of which are compensated for by software. And there is no safety culture in software (see Travis, 2019, for an interesting analysis on this).

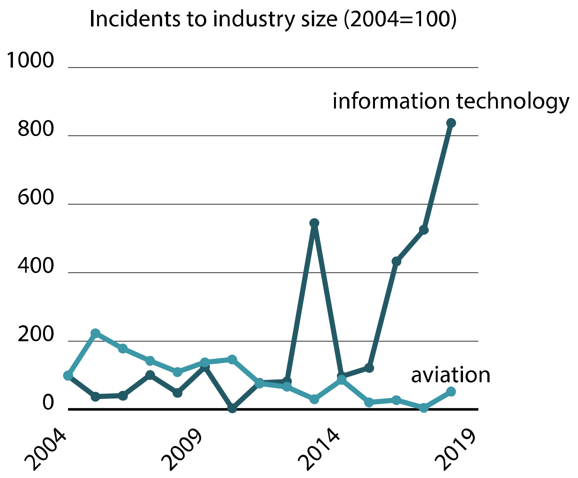
One can look at the statistics. Since the early 1970s, the number of casualties in civil aviation has consistently decreased, despite the industry's growth, especially in emerging markets (data: Aviation Safety, 2019).



Cybersecurity tells a different story. Using data breaches as an indication for system compromises by hacking or negligence, the numbers have been rising since 2004 and have literally exploded during the past three years (data: author's analysis from Information is Beautiful, 2019a).



Obviously, information technology has grown at an even more spectacular rate than commercial aviation has. But at the same time, airline accidents do not go unnoticed, whereas there is a substantial number of unreported cases in cyberattacks. Looking at the ratio of incidents to industry size for the past fifteen years and indexing at 2004, the security problem of information technology is undeniable (reference market size data: Internet World Stats, World Bank, 2019).



Vendor accountability

A safety culture goes hand in hand with vendor accountability. In April 2017, The Economist pinpointed it:

"The software industry has for decades disclaimed liability for the harm when its products go wrong."

The cybersecurity problem cannot be resolved with more technical wizardry and claiming that users should be more careful and protect themselves. It will need economic tools to set the right incentives for companies to take reasonable steps to make their products secure. In consequence, manufacturers will need to slow down a bit and work on the security basics, before adding ever more features. "Firms should recognise that, if the courts do not force the liability issue, public opinion will" (Economist, 2017a).

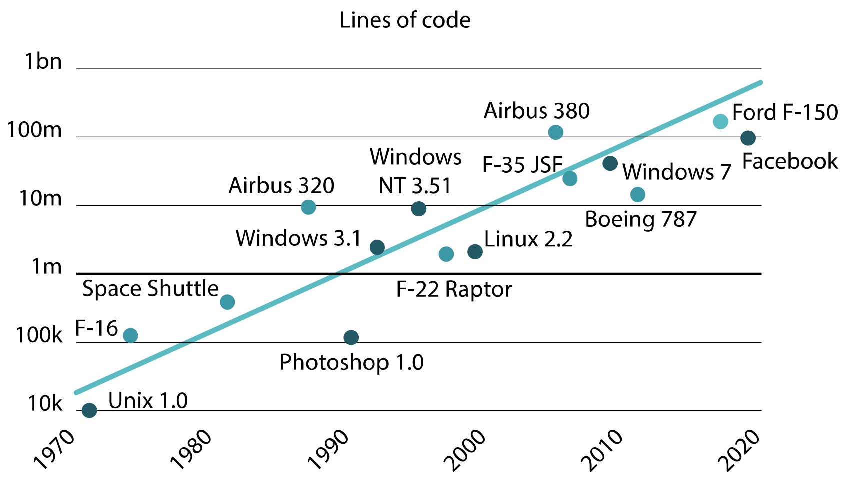
Niels Ferguson, Bruce Schneier and Tadayoshi Kohno are amongst the world's most renowned computer security experts. In their milestone book *Cryptography Engineering* (2010), they, too, criticise the software industry's lax attitude, comparing it to the automotive industry, where a defect leads to a manufacturer recalling the affected vehicles, whereas software companies just waive all liability. They go on:

"We find it instructive to look at the best engineering quality control system in the world:
the airline industry." (p. 119)

So, what does aeronautical engineering teach us? How can software be developed and be deployed with acceptable confidence that it is sufficiently secure — despite its obvious complexity?

Software complexity

Oh, right – complexity. The increase in software complexity over the past decades has been dramatic, the actual level of which is best understood by looking at the number of lines of code that a typical product contains (data: NASA, 2009; Wiels et al, 2012; Economist, 2017b; Information is Beautiful, 2019b).



On a logarithmic scale, the graph provides for some great insights. First, each era seems to have a natural code size; it inflates by an order of magnitude every eleven years. Second, code sizes are remarkably similar despite belonging to such different products as computer operating systems, aircraft, or even a pick-up truck. The year-2000 developments F-22 and Linux 2.2 required around 2m lines of code; their 2010 successors F-35 and Windows 7 needed ten to fifteen times as many.

It seems that code size is less driven by the need for functionality than by the possibilities that hardware and development environments offer. In the 1990s, a development team could handle about a million lines of code; tooling allowed them to juggle with twenty million lines around 2010 and by now, the 100m mark has been passed. You need not be a fundamentalist system administrator to question the need for such code bases. Does a pick-up truck really need 150m lines of code, ten times as much as a Boeing 787? (Hint: do not forget that the Ford can drive backwards without the help of a tow truck.)

Requirements and documentation

But let's return to the safety culture. The Airbus 320's 10m lines of code result in an incredibly safe and reliable product, something only few people would say about Windows 3.1, even if it has only 2.5m lines of code. The above diagram with code sizes disproves any correlation between total code complexity and system reliability. One *can* develop a highly complex system and yet make it safe.

The key is the classification of code into categories of criticality. Depending on the consequences when thing go awry, code must be developed with a different mindset from the very beginning of writing down requirements — that is, for many modern-day pieces of software, requirements need to be written down at all. For more critical code, one cannot and must not take some well-known libraries, use their functions without any sanitisation of inputs and outputs, and link the whole library into the product — also adding functionality that no-one wants or even knows about.

There is a perceived conflict of interest between modern-day software development and requirements engineering. There are evangelists of agile methods who turn green if one mentions the word requirements. Requirements are so waterfall.

The argument goes like this: Experience shows that software tends to be late, over budget, and expensive to maintain. The root cause is the system’s complexity, making it impossible to do it right from the beginning; the process must be iterative. User feedback is crucial all the way along the development path.

Sounds familiar? It does — but the argument dates back to 1968. These were exactly the points being discussed at the NATO Software Engineering Conference (NATO, 1969). A solution was developed and first described by Royce (1970). It aims to avoid costly architecture redesign and code refactoring by carefully looking at the user requirements. The name by which it became known: the waterfall model.

Agile methods do not mean to start to code without a plan and to improvise on the code until it works. It will work, and probably even so in most of the normal use cases. But it will definitely fail if something unforeseen happens. Something like a wormable vulnerability of the likes of EternalBlue. Agile methods *do* mean to iterate through high-level and low-level requirements and code fragments in parallel, in order to arrive at a consistent set of both, even if the full scope could not be overseen at the beginning. The irony in it: all of the concepts were already there by the end of the 1960s. Evolutionary prototyping, throw-away prototyping, continuous end-user interaction: all there (see for example Comer, 1990).

Now, this is a good thing. The software certification processes that brought us the safety of aviation are based on the waterfall model, including requirements, design, coding, and integration activities, and formal documentation. But is does not prescribe the actual life cycle model. Nowhere, it demands that requirements are finished to remain untouched before coding starts. The model allows for iterative development — moreover, it even encourages to do so. And then it forces one to document new insights, new designs, new concepts, new requirements. It will thus enforce multiple revisions of milestone reviews and quality gates. And isn’t that exactly what agile methods are trying to teach us?

“Some questions are easy to answer if you can find the person that actually implemented the code (…); it becomes much more difficult when that person is no longer with the company and there is no documentation available.” (Higaki, p. 87)

Certification criteria

So, there we have it. Software may have grown in size and complexity, but the problems have been around for more than half a century — thus their causes do not lie in complexity. Large, complex systems can be developed for safety and reliability, and — as aviation demonstrates — successful certification is possible. The key is to categorise and classify parts of the full system according to probability and the consequences of failure — and to adapt the level of test, evaluation and scrutiny accordingly. Iterative development is inevitable — but not a solution to the challenges of complexity. And documentation is mandatory.

In aviation, all of this is prescribed by the *Software Considerations in Airborne Systems and Equipment Certification* (DO-178C, 2011). The categories of criticality are named 'software levels', ranging from A for the most critical to E for the least critical. Objectives on the software development process, its tooling, testing, and traceability are listed in a series of tables, containing between 26 (for level D) and 70 (for level A) items.

The equivalent in cybersecurity are the *Common Criteria for Information Technology Security Evaluation* (CC, 2017), jointly developed by twelve governmental agencies from different countries. It introduces 'evaluation assurance levels', ranging from 1 for systems that are only functionally tested to 7 for those that are formally verified design and tested. They differ in meaning from DO-178C's software levels, but the impact on evaluation of the product is highly similar: with increasing criticality, there are more criteria to be met.

The DO-178C document is not very long, considering its importance and impact. After all, it only lists objectives and does not prescribe a detailed methodology. The downside is that DO-178C cannot be used as a reference guide. Organisations will need to learn how to work with it and develop their own processes that meet all the objectives. Yet, the aerospace industry is large. There are numerous companies that are proficient in developing airborne software, from small component suppliers to large platform integrators.

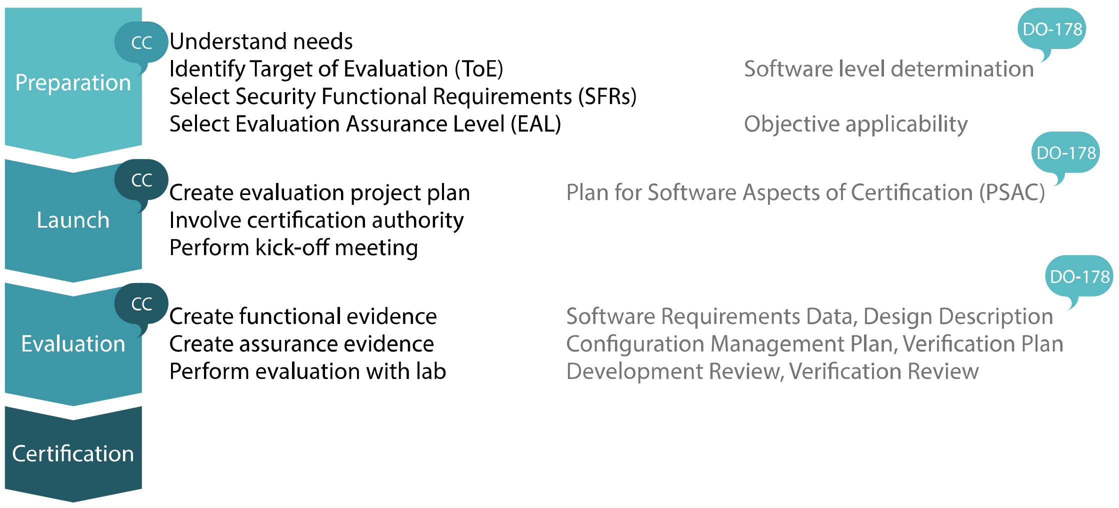
The CC are a bit more elaborate, but like DO-178C, they are not a tutorial either. The document describes what to demonstrate — not how to get there. And there is by far not as much experience as the aerospace industry has with DO-178C and its predecessors. The CC website lists all certified products to date: a total of 2,575 in August 2019, almost half of which are from the single product category of smart cards and the corresponding readers. To this day, CC-experienced companies are hard to find.

Making cyber secure

The similarities between airborne system certification and information technology security certification allow for intuitive application of procedures and practices from aerospace to the cybersecurity problem. A good Common-Criteria process starts with internal preparation. It is important to understand the need for certification, and to identify the parts of the product that should be evaluated (the Target of Evaluation). In parallel, the security functional requirements (SFRs) for the product should be identified, and the level of evaluation should be chosen, implying a series of security assurance requirements (SARs). In all of this, experience with the evaluation of failure condition categories and software levels from DO-178C — together with its certification objectives — can be applied.

When the process is formally started, an evaluation lab — an independent third party that does the actual security evaluation and produces a report for the national authority — and the certification authority itself must be involved, highly similar to the involvement of the airworthiness authorities in the aviation case. The Plan for Software Aspects of Certification (PSAC) from DO-178C can serve as a nice template for an evaluation plan in cybersecurity.

Further down the road, evidence must be produced to demonstrate how each of the functional and assurance requirements is met. Software requirements data, design descriptions, configuration management plans and verification plans again provide the framework for what to document, and how. Depending on feedback and questions from the evaluation lab, some rework will be necessary until the product is actually certified by the authorising body.


Preparation
CC
Understand needs
Identify Target of Evaluation (ToE)
Select Security Functional Requirements (SFRs)
Select Evaluation Assurance Level (EAL)
DO-178
Software level determination
Objective applicability
Launch
CC
Create evaluation project plan
Involve certification authority
Perform kick-off meeting
DO-178
Plan for Software Aspects of Certification (PSAC)
Evaluation
CC
Create functional evidence
Create assurance evidence
Perform evaluation with lab
DO-178
Software Requirements Data, Design Description
Configuration Management Plan, Verification Plan
Development Review, Verification Review
Certification

Whether the formal process is worth the effort is difficult to say upfront — that is why it all starts with understanding the need for certification. It is very well possible — and may in many cases be more rational — to apply the Common-Criteria processes, leverage the DO-178C experience, but refrain from getting the actual certification.

Stop history from repeating

Because: an information technology product does not become more secure when it is evaluated according to CC assurance level 7, than when doing so for level 1 or 2 — or not evaluating at all. Common criteria certification is optional and — combined with the vendor's freedom to decide which parts of a product are to be evaluated and at which level — basically a marketing tool. But despite being mandatory, airborne systems also do not become safer from certification according to software level A. It only indicates that the system was deemed more critical and that more effort went into the evaluation process. The evaluation and certification processes help to ask the right questions, to make sure nothing gets forgotten. In the end, the mindset and actual way of working of the developers, driven by the habits and expectations of the organisation, is what counts. This is what makes for a safety culture.

Aviation has one. The many companies that make for the industry are in an excellent position to apply their experience in creating more secure information technology products. Life cycle models that include system requirements analysis and maintenance, design documentation, high- and low-level requirement engineering, and requirement-to-code traceability, are perfectly suited to create common-criteria documentation on the fly. And to stop the history of late, over-budget, poorly maintainable and insecure software from repeating. To stop screens from going black. With or without a certificate.

How ACTRANS can help

ACTRANS combines decades of experience in the European aerospace industry with current-day knowledge from the cybersecurity industry. We can help companies to improve their products' cybersecurity by supporting them along the way of understanding Common Criteria and developing the right safety culture. We contribute in identifying a security target and preparing the business case for an evaluation process. If formal certification is worthwhile, ACTRANS can find the right evaluation lab, coach in-house employees on the process, and support in achieving effective reviews.

References

Aviation Safety (2019). *Airliner fatalities by period*. <https://aviation-safety.net/statistics/period/stats.php>.

CC (2017). *Common Criteria for Information Technology Security Evaluation 3.1* (5). <https://www.commoncriteriaportal.org/cc/>.

Comer, E.R. (1990). "Alternative software life cycle models" in Anderson, Ch. and Dorfman, M. (eds): *Aerospace Software Engineering*. Washington DC: AIAA, pp. 69-86.

DO-178C (2011). *Software Considerations in Airborne Systems and Equipment Certification*. Washington DC: RTCA.

Economist (2017a). "How to manage the computer-security threat" (printed as "The myth of cyber-security") in: *The Economist April 8th 2017*. <https://www.economist.com/leaders/2017/04/08/how-to-manage-the-computer-security-threat>.

Economist (2017b). "Computer security is broken from top to bottom" (printed as "Why everything is hackable") in: *The Economist April 8th 2017*. <https://www.economist.com/science-and-technology/2017/04/08/computer-security-is-broken-from-top-to-bottom>.

Ferguson, N., Schneier, B. and Kohno, T. (2010). *Cryptography Engineering*. Indianapolis: Wiley, pp. 115–119.

Greenberg, A. (2018). "The untold story of NotPetya, the most devastating cyberattack in history" in: *Wired August 22nd 2018*. <https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world>.

Higaki, W.H. (2010). *Successful Common Criteria Evaluations*. Higaki.

Information is Beautiful (2019a). *Data breaches (public)*. <https://docs.google.com/spreadsheets/d/1Je-YUdnhjQJO_13r8iTeRxpU2pBKuV6RVRHoYCgiMfg>.

Information is Beautiful (2019b). *Lines of code*. <https://docs.google.com/spreadsheets/d/1s9u0uprmuJvwR2fkRqxJ4W5Wfomimmk9pwGTK4Dn_UI>.

Internet World Stats (2019). *Internet growth statistics*. <https://www.internetworldstats.com/emarketing.htm>.

NASA (2009). *Study on Flight Software Complexity*. Dvorak, D.L. (ed). <https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf>.

NATO (1969). Software Engineering. Naur, P. and Randell, B. (eds). <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf>.

Royce, W.W. (1970). "Managing the development of large software systems" in: *Proceedings of the IEEE WESCON 1970*. <http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf>.

Travis, G. (2019). "How the Boeing 737 Max disaster looks to a software developer" in: *IEEE Spectrum April 18th, 2019*. <https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer>.

Wiels, V., Delmas, R., Doose, D., Garoche, P.-L., Cazin, J. and Durrieu, G. (2012). "Formal verification of critical aerospace software" in: *Aerospace Lab Journal 4* (10). <http://www.aerospacelab-journal.org/sites/www.aerospacelab-journal.org/files/AL04-10_1.pdf>.

World Bank (2019). "Air transport, registered carrier departures worldwide" in: *Data*. <https://data.worldbank.org/indicator/IS.AIR.DPRT>.