Evaluating Automated Software Verification Tools

Christian R. Prause
DLR Raumfahrtmanagement
Königswinterer Straße 522-524
53227 Bonn, Germany

Rainer Gerlich
Dr. Rainer Gerlich
BSSE System and Software Engineering
Immenstaad, Germany

Ralf Gerlich
Dr. Rainer Gerlich
BSSE System and Software Engineering
Immenstaad, Germany

***Abstract*—Automated software verification tools support developers in detecting faults that may lead to runtime errors. A fault in critical software that slips into the field, e.g., into a spacecraft, may have fatal consequences. However, there is an enormous variety of free and commercial tools available. Suppliers and customers of software need to have a clear understanding what tools suit the needs and expectations in their domain. We selected six tools (Polyspace, QA C, Klocwork, and others) and applied them to real-world spacecraft software. We collected reports from all the tools and manually verified whether they were justified. In particular, we clocked the time needed to confirm or disprove each report. The result is a profile of true and false positive and negative reports for each tool. We investigate questions regarding effectiveness and efficiency of different tools and their combinations, what the best tool is, if it makes sense at all to apply automated software verification to well-tested software, and whether tools with many or few reports are preferable.**

I. Introduction

Twenty years ago, the first Ariane 5 launcher rocket self-destructed 40 seconds after initiation of the flight sequence. The active as well as the redundant on-board computers had shut down, all suffering from the same software failure. Software plays an ever more important role in spacecraft (e.g., satellites), which are often one-of-a-kind devices with custom-built peripherals [16]. And again and again does software cause critical failures, e.g., the recent loss of the 286 million USD Hitomi satellite [37], which is just one example of spaceflight's history of expensive software failures (cf. [18], [21], [31]). Guarding against such failures typically includes a multitude of measures, ranging first and foremost from testing and validation, over various safety and dependability analyses, to standardization, process control and improvement, and management considerations. One important activity among these is verification of source code using automated software verification (ASV) tools. When software development is contracted, suppliers and customers need to know what results to expect from verification processes using these tools.

The ECSS (European Coordination for Space Standardization) standard for software engineering states: "The supplier shall verify source code robustness (e.g. resource sharing, division by zero, pointers, run-time errors). AIM: use static analysis for the errors that are difficult to detect at run-time." [1] Product assurance requirements for the German national space program additionally demand to perform static analysis to assure correct control and data flow, internal consistency between units (number and types of parameters passed to functions including function pointers), the absence of non-deterministic behavior, data corruption and security breaches, and, in particular, the absence of run-time errors like divisions by zero, square roots of a negative numbers, overflows and underflows, out-of-bounds array and pointer accesses, illegal type conversions, and reading of non-initialized data.

Disputes easily entail between customers and suppliers about which tools support achieving the necessary quality of code verification. Both parties need to know how well verification tools can detect faults. Emanuelsson and Nilsson note that "even when the tools seemingly provide the same functionality (e.g. detection of dereferencing of null pointers) the underlying technology is often not comparable; each tool typically finds defects which are not found by any of the other tools." [11] The choice of tools has an effect on the effectiveness (e.g., risk that faults slip into the field, and types of faults detected) and efficiency (e.g., expected percentage of false positive reports and developer time spent for analyzing these reports). On the one hand, companies and government agencies in various domains and countries have this same need. On the other hand, reliable and detailed information is difficult to come by. Consequently, organizations evaluate the same verification tools for similar uses over and over again [8].

We therefore devised a method to evaluate ASV tools (see Section II) and applied six tools to spacecraft software. Experts manually validated the generated reports for correctness. We then consolidated the list, and analyzed data statistically (see Section III) to answer the following questions:

1) Is it justified to apply ASV to already qualified software?
2) What is the best ASV tool available?
3) Are there significant differences between tools' capabilities?
4) Does longer analysis runtime mean less reports or better results?
5) Are tools that issue more reports less cost-efficient than tools that report fewer ones?
6) Is it effective and efficient to apply more than one tool?
7) Would a simpler evaluation (e.g., counting reports) lead to comparable results?

Section IV discusses our answers to the above questions based on data from real-world software, and reviews of experienced verification engineers. Section V presents related work. Finally, we conclude (Section VI).

Prause, C.R., Gerlich, R., Gerlich, R.: Evaluating Automated Software Verification Tools. In: 11th IEEE Conference on Software Testing, Validation and Verification. 2018
@inproceedings{prause:evaluatingASVTools, title = {Evaluating Automated Software Verification Tools}, author = {Christian R. Prause and Rainer Gerlich and Ralf Gerlich}, year = {2018}, booktitle = {11^{th} IEEE Conference on Software Testing, Validation and Verification. Industry Track}, series = {ICST}, publisher = {IEEE Computer Society}, }

II. Materials and Methods

This section provides details on the study implementation.

A. Terminology

To avoid misunderstandings in the following parts, we first present our definitions of several terms.

1) Defect, fault, error, failure: A *defect* is any problem in software or its (source) code that affects either internal/structural (e.g., code smells, violations of code layout rules) or external/functional quality. A *fault* is a human mistake that manifests in the code and that rests latent until executed and activated; i.e., a functional defect. An *error* is the erroneous state of the system that may result under particular circumstances from activating a fault. The error may lead to a software or system *failure*: unexpected or undesired behavior of the system [2]. For example, a program might crash (Ariane V), or continue to operate but in an undesired way (Hitomi).

In this paper, we only consider faults, and exclude defects without functional effects. Albeit, in general, taking care of non-functional defects is justified: The MISRA-C [24] coding rules for critical software, for example, restrict use of the C language to safer subsets, to avoid certain potential problems. Under the umbrella term of *technical debt*, there is huge interest in these kinds of defects.

2) Automated software verification: The ECSS defines software verification as the process to confirm that a "product is designed and produced according to its specifications" [3]. Among other verification activities, the "supplier shall verify source code robustness (e.g. resource sharing, division by zero, pointers, run-time errors)" by means of, e.g., the "use [of] static analysis for the errors that are difficult to detect at run-time" [1], testing, and other methods [3].

A conventional distinction is made between *static* and *dynamic* analysis. Static analysis detects potential faults by analyzing source or binary code without executing it, e.g., coding rule checking or abstract interpretation. Dynamic analysis (or testing) executes software to identify unexpected or undesired behavior; for instance, unit testing, or test data generation. Young and Taylor [39] propose to discriminate between pessimistically inaccurate *folding* that is *sound* because it does not miss a fault, and optimistically inaccurate *sampling* that is *complete* in its analysis of fault candidates because every fault reported truly is one (cf. [7]).

We do not preclude either of these techniques from our study. Instead, the decisive criterion is whether the tool will work without requiring much specific prior knowledge about the software under test. Hence the term *automated* verification. This implies an implicit definition of possible undesired behavior like illegal memory accesses, division by zero, overflows, etc. Therefore, most static analysis and randomized testing tools qualify; but unit testing based on test scripts does not, as it requires prior knowledge in form of the scripts.

3) Signal detection theory: In signal detection theory, a signal (here: software fault) is either present or not present. A detecting system can either determine that the signal is present or not, i.e., an ASV tool either reports a fault or it does not do so. If an ASV tool correctly reports that there is a fault, or does correctly not report a fault, we speak of *true positive* (TP) and *true negative* (TN), respectively. Likewise, if an ASV tool incorrectly reports a fault although there is none, or reports none although there is one, we call these *false positive* (FP) and *false negative* (FN), respectively. From the resulting 2×2-cell confusion matrix, we can derive the quality criteria *sensitivity* (S) and *precision* (P) for each tool as

$$S = \frac{TP}{TP + FN} \qquad P = \frac{TP}{TP + FP}$$

where sensitivity is the ratio of actual faults detected, and precision the ratio of correct reports. A typical quality measure derived from the confusion matrix is the Matthews correlation coefficient. However, the true negatives would dominate the other three cells. Instead, the Jaccard similarity coefficient, which is a measure of the overlap between actual and reported faults, is a more suitable quality measure in this case because it is independent of true negatives:

$$J = \frac{TP}{FP + FN + TP}$$

B. Material

This section presents the materials for our study.

1) Software under test: In order to study the ASV tools, we needed source code to which they can be applied. Several detailed evaluations of ASV tools use specifically crafted pieces of source code (see Section V). Shiraishi et al. [33] give advice on constructing such a test suite.

Instead of creating a laboratory environment, we decided to base our analysis on real-world software from our domain. The decision is basically a trade-off between *internal validity* (i.e., control about the experimental setting) and *external validity* (i.e., representativeness and realism). Risks/downsides of using real-world software are that the true number of faults are mostly unknown (false negatives), laborious analyses are necessary to decide whether a report is a true or false positive, and that one has no control over the distributions of fault types; in particular, some kinds of faults may be present several times, while others are not present at all.

Benefits of using real-world software are increased realism because we have inherently natural fault distributions, and that ASV tools do not analyze small, isolated code snippets but have the full program context available. One tool in our test grouped together reports that originated from the same fault; had we only used single-fault snippets, this usability feature would not have become visible. Also, the process of first collecting reports and then deciding whether a report is justified better resembles real-world usage. Furthermore, we can clock analysis times, which vary considerably for reports.

SuT (pseudonym, Software under Test) is an on-board software/middleware for controlling spacecraft and its peripherals. It is classified as ECSS criticality class B, i.e., if not executed, or not executed correctly, can lead to the loss of a mission. The development contract was awarded to a major commercial space system company, and supported by

a research institute. The analyzed software package contains 610 functions in $\approx$ 42000 lines of code. At the time of this study, the software had passed its *qualification review*, i.e., the last milestone review before acceptance: It was fully validated against its requirements baseline, and verification activities were completed (cf. [1]). It is said to be "qualified". Coverage from automated unit tests was about 90%.

2) Selection of ASV tools: Due to high demand for support of software verification, a vast variety of commercial and free ASV tools exists. Out of this variety, we selected six tools that we deemed particularly relevant.

a) Code Prover (Polyspace): is a sound static analysis tool, i.e., it claims to be able to prove the absence of certain types of runtime errors. Polyspace was developed from a science prototype in the aftermath of the Ariane 5 disaster [22]. It is therefore a natural choice for our study. Technically, it builds on abstract interpretation based on complex polyhedra to determine the possible values of variables of a program. It highlights code according to a traffic light metaphor of green (proven to be correct), red (proven to cause a failure), or orange (undecided). Among contractors, it is particularly feared for its license cost, steep learning curve, and orange messages which can amount for up to 20% of the code (see [10], [11], [30]). For an average flight software, this can mean several thousand orange reports, which must be checked.

b) Bug Finder (Polyspace): is offered as a light-weight version of the Code Prover. It aims to be complete (i.e., make no false reports) and has a shorter execution time so that developers can use it on their desktop computer while developing, allowing them to fix easily detectable faults immediately. It supports coding rules checking. However, Bug Finder cannot to prove the absence of runtime errors.

c) QA C: has a history of more than twenty years, and has broad popularity in automotive industry. As many other tools, it started out as a coding rule checker but has since seen several improvements. It now has support for advanced data flow analysis. QA C has far-reaching support for coding rules (in particular, MISRA-C [24]), which can easily cause thousands of complaints if other coding standards were used during development. However, as we do not want to check coding rules, we filtered out any MISRA-C complaints.

d) Klocwork: focuses on finding bugs and fixing security flaws. The tool is meant to be closely integrated into the developer's desktop, to be as close as possible to where code is written, modified, tested and reviewed. This enables finding problems at the earliest point in the process, resulting in fewer failed tests and fewer impacts on cost and schedule.

e) DCRTT: is an in-house product of a small software verification company that specializes on critical embedded software and offers verification services to aerospace enterprises and agencies. DCRTT employs randomized testing, executing functions of the software under test with random data, to gather information that hints at undesired behavior.

f) gcc: is the GNU C compiler; being a compiler, it is at the forefront of static analysis. One does not naturally expect highly sophisticated analysis results from a compiler. However, the gcc is popular as compiler among space software companies. In our study, gcc is used with the `-Wall` option.

3) DeWitt clauses and tool anonymization: DeWitt clauses are today part of many end user license agreements (see [19] for examples). In essence, they forbid the software licensee to publish the results of testing or benchmarking of the licensed software. Klass and Burger [19] judge this as wasteful because developers will have to spend precious time on re-evaluating the same tools, while the quality of the evaluations themselves will, in many cases, remain rather shallow.

The so-called Toyota-study of static analysis tools [32] gained quite some attention among practitioners in 2015. However, it has since been withdrawn from IEEE Explore. John Regehr, having linked to a copy of the study in his blog, writes: "Whoops – it looks like Coverity/Synopsys has a DeWitt clause in their EULA [end-user license agreement] and based on this, they sent me a cease and desist notice."[1] According to German law, everybody has the right to observe, investigate and test the behavior of a program to understand its ideas and principles, and these rights may not be restricted through contracts. Respective clauses would be legally void. Yet the publisher of this paper is IEEE and located in the US. While Klass and Burger [19] note that a DeWitt clause will probably not hold in court, it has not been tested yet. We therefore anonymize tools to ToolA, ..., ToolF to separate names from results to be on the safe side. Furthermore, we want to stress that our goal is not to recommend a single best tool (this information might already be outdated with the next version of an ASV tool) but to present a study methodology, explain difficulties of evaluating ASV tools, and show how diverse the ecosystem of ASV tools is.

C. Method

Basically, the evaluation method is as follows:

1) Apply each ASV tool to the SuT, and collect the reports
2) Consolidate the reports into a single set, while noting which tool reported what
3) Validate each report to decide whether it is true or false positive; clock and record analysis time
4) Perform data analysis and evaluation

Next, we present intricacies and details of this process.

1) Apply ASV tools: To collect the tool reports, we used each tool on the software under test.

a) Configuration of tools: Configuring ASV tools turned out to be non-trivial in most cases. We ran several tools multiple times, fiddling around with and optimizing configuration options. By doing several iterations, we controlled the risk of unfortunate/unfair configurations.

b) Fault type catalog: Different tools give different names to the same faults: "Unreachable Code" may be called {"Dead code", "Unreachable code", "wasAlwaysFalse", "INVARIANT_CONDITION.UNREACH", "Invariant operations"}, or "Array Index Out-of-Bounds" may be {"Out of

[1] https://blog.regehr.org/archives/1217 Accessed 2017-08-21

bounds array index", "Excp", "ABV.GENERAL", "2844 Arrays"}. We created a fault type catalog [12] to handle naming issues, describe kinds of faults, show examples and give faults unambiguous names throughout our evaluation.

c) Criticality of reports: Some ASV tools issue criticality levels alongside the reports that can be used for filtering. We use our own levels that are documented in the fault type catalog. The least critical class of faults are non-functional defects like style violations. We do not deal with these in our study. Next are warnings that indicate a possibly unintended operation in the source code which may (but not necessarily does) manifest as an error. It might signal a problem or deviating developer intentions but is not a fault in the first place (e.g.: Arithmetic Operation on NULL Pointer, Multiple return paths, Invariant Condition, Unreachable Code and Unused Result). Critical faults impact the correctness of system operations if activated, i.e., resulting in error or failure (e.g.: Array Index Out-of-Bounds, Dereference of NULL-Pointer, Non-terminating Loop, (Possible) Recursion and Undefined Result of Arithmetic Operation).

d) Data export: Several tools came with their own GUI. Exporting machine-readable lists of reports is not the normal way of working with the tools. Furthermore, line numbers are not included in exports by default. Of course, this complicates automated processing of report data. We had to obtain special, normally unavailable licenses that allowed to export line numbers. Inquiring why, a vendor told us that if they allowed such export, much fewer licenses would suffice to check the same amount of software, because a few analysts could generate the reports and then distribute them to developers. There would be no need to continuously use the tool; what they consider a misuse. In our experience, all vendors we had to contact had no reservations against providing the respective licenses after understanding why we needed it. When conducting a similar study, one should check in advance with the vendors that they are willing to provide machine-readable result data.

2) Consolidate reports: We collected all the reports generated by the ASV tools in a single table, removing duplicate reports by different tools regarding the same suspected fault.

a) Manual merging: We intended to automate most of the consolidation work, exploiting location (file + line number) and fault type information. We anticipated only some manual work for harmonization. However, while consolidation did not require in-depth analyses of individual reports, matching related reports caused significant manual effort. For example, tools reported slightly different locations for the same report.

b) Analyzed Subset: Due to the enormous effort for analysis, we selected a subset of 60 functions ($\approx 10\%$ of functions and lines of code) for further analysis. We selected 30 functions randomly and 30 ones with the highest numbers of reports. Interestingly, the average number of reports per line of code was similar in both sets. Functions in the second set were simply longer.

c) Reporting of consecutive faults/grouping: Sometimes several consecutive reports may have the same single fault as their origin, and can be grouped. For example, imagine an array of length n, where index $n + 5$ is accessed before index $n + 3$ is accessed. Both accesses may cause an error because the array is too short/index too high. When executing the program, accessing $n+3$ could not lead to a failure because accessing $n + 5$ would have done so before. We call $n + 3$ a consecutive fault of $n + 5$, and a tool may decide to not issue a separate (unnecessary) report in this case (e.g., for usability reasons). However, if consecutive faults are not dealt with separately, either the tool would have a false negative (for not reporting), or other tools would have a false positive (for reporting a non-fault), which would both be unfair. Hence, we checked for consecutive faults and tracked information about them. Some tools also provided grouping information.

3) Validate reports: Experts judged every report in the consolidated list to determine whether it was true or false positive. We clocked and recorded the time (in minutes) needed for each report separately.

a) With- and without-context view assessment: Whether a fault is present depends on the values of variables and parameters at the location of the fault. The values are typically constrained by the program execution before reaching the respective location (call context). Thus not all values are possible, and, consequently, an error may be possible in one context, and impossible in another one. A report is then a true or false positive depending on whether context is considered. The with-context view is more precise, however, only as long as the context does not change. Library functions, for example, may be used in changing contexts. And must then be re-analyzed every time. The without-context view is more sensitive but causes more false positive reports.

The experts therefore judged every report twice: once assuming the without-context view — which is usually easier and faster — and once assuming the with-context view. In the with-context case, judgments considered the whole call tree up to the main entry point if necessary. Both analysis times were recorded: first the without-context time, and then separately the additional time for with-context analysis.

4) Data analysis and evaluation: Finally, statistical data analysis and evaluation commenced.

a) Data format: Our fault database has one row for each fault, with columns as explained in Table I. For easier handling, the fault database was created in Excel. When it was finished, it was imported into R for further statistical analysis.

b) Tool Vendor De-Briefing: When data collection was finished, we conducted de-briefings with the tool vendors. We wanted to re-check again whether we made a blunder when applying tools and potentially iterate data gathering, to share results regarding tools with their respective vendors (EULAs prohibited sharing detailed results across vendors), and because some vendors were very interested in feedback.

The de-briefings confirmed that we did not make any major mistakes when applying the tools. In-depth technical discussions came about with the development team of one vendor about a handful of false negative reports that, according to the tool's handbook, should have been detected. It turned out that the handbook had to be updated. Regarding feedback,

TABLE I
COLUMNS OF THE REPORT DATABASE

Field	Description
ID	Unique identifier for the suspected fault; i.e., its ID in the consolidated list
file	Path and file name of the file where the report is found
function	Name of function containing the report
line	Source line number of report
type	Fault type according to our catalog
description	A human-readable description of report that justifies its type classification
implied by	If report is a consecutive fault, then the other one's ID (cf. grouping)
decision w/o context	TRUE if report is true positive without considering context, FALSE otherwise
analysis time	Minutes spent for analyzing report without considering context.
justification	Human-readable explanation for above decision
[decision with context]	Above three fields repeated for the "with-context" case
[found by ToolA...F]	For each ASV tool, whether or not it issued this report

interest of the vendors was very mixed. It ranged from the in-depth discussions to polite disinterest of one vendor that mentioned that their users had a different use case.

c) Handling of consecutive faults: Recall that consecutive faults cannot cause an error because a system would already be in an erroneous state due to the predecessor. In order to normalize data, we decided to ignore reports for consecutive faults because they disadvantage tools that do not report them but only the predecessor. However, a tool that only reports the consecutive faults but not predecessor would then be disadvantaged. So, analysis first iterated through all "impliedBy" relations in the data: if a tool reported a consecutive fault but not the predecessor, we assumed that it had, instead, reported the predecessor. Nine reports were affected. We excluded 147 critical and 83 warning consecutive faults from further analysis (between 30-40% for most tools, cf. Table II). The consecutive faults caused additional effort in the study. We had to manually decide whether — and, if yes, in what way — they were related to which other faults. In the real world, consecutive faults are usually fixed implicitly when the predecessor fault is fixed. So they will not show up again when the ASV tool is rerun, need not be fixed explicitly, and should cause only negligible effort (as long as the tool is used iteratively, and in parallel to development).

d) Combining with- and without-context views: In order to simplify results, we integrated both cases into one view: Results are primarily based on the decisions for the with-context case (see [13] for more details on the two cases). However, we treat critical reports that are false positive in the with-context case and true positive in the without context case as true positive but warning reports. The reasoning is that the pragmatic scenario for ASV is to ensure that software is free of critical faults as cheap as possible. Still, critical without-context reports are potential future maintenance problems very much like with-context warnings. So they are worthwhile being dealt with as well. This modification affected 16 reports.

III. RESULTS

From the consolidated reports, we extracted information about the six ASV tools.

a) Comparison of key figures: Figure II presents overall per tool statistics. Six columns present the numbers for individual tools, whereas "Overall" presents an overall table for all tools, which is computed as whatever is more appropriate for the row. It can be the sum of the tool columns (e.g., unique contribution), using overall lists (e.g., sensitivity and precision ⑦), (weighted) averages (e.g., rows Ⓐ), or multiple linear regression (e.g., rows Ⓕ).

The first row of the table ① lists the runtime of the tools. There are two clusters of tools: one with tools with rather short run times, and one with significantly longer times. ② shows the percentage of true positive reports in the without-context view that become false positive in the with-context view. Again, there are two groups of tools: 0% and $> 0\%$. These two groups are also visible in ③, which shows the total number of critical reports issued by the tool. ToolD has issued only one report (③ and ④). It was probably used by the developers during development and its reports immediately addressed, or it did not issue many reports at all. The unique contribution ⑤ is the number of critical faults detected exclusively by the tool. There were 39 faults that only one single tool detected; more than half of them came from ToolB. ⑥ shows the ratio of consecutive faults among all of the tool's reports. ⑦ reports the sensitivity and precision of reports split by warning and critical, and overall. Unsurprisingly, the two groups show in the sensitivity for critical reports. ⑧ is the total time (in minutes) the experts wasted judging false positive reports by the tool. The tools that score high on detecting critical faults, also cause the highest efforts for analysis of false positive reports. The effect carries forward in overall total analysis times, and the derived time needed per source line of code ⑨. Note that the Overall times for using all tools are less than the sums of all tools' times because of overlaps between tool reports. A clear distinction between average times (in minutes) to find a warning or critical true positive Ⓐ is not obvious. What can be seen, however, is that the overall average time necessary to find a critical true positive is much higher than for warning true positives. Minimum analysis times per report are often 0 minutes because analysis times of less than 30 seconds were rounded down. The Jaccard similarity coefficient in Ⓑ measures the overlap of the tool's reports with the list of actual faults, i.e., what it should have reported. Once more, ToolB, ToolC, and ToolF give better results than the other three tools. Yet ToolB and ToolC appear to be more effective for critical faults and not so much for warnings, while ToolF

TABLE II
COMPARISON OF SEVERAL KEY FIGURES OF TOOLS

		Overall	ToolA	ToolB	ToolC	ToolD	ToolE	ToolF
①	Tool runtime (minutes)	-	10	300	600	15	3	5
②	True positive $\xrightarrow{\text{with context}}$ false positive	11%	0%	13%	20%	0%	0%	10%
③	Total number of critical reports	86	8	73	48	1	11	58
④	Total number of warnings reported	184	13	14	71	1	17	96
④	Ratio of critical reports out of all reports	32%	38%	72%	29%	50%	39%	31%
⑤	Unique contribution (critical)	39	1	23	10	0	2	3
⑥	Consecutive fault ratio	$\approx 50\%$	32%	48%	38%	33%	72%	37%
⑦	Sensitivity / precision (total) in %	100 / 78	10 / 90	37 / 77	38 / 59	1 / 100	13 / 89	69 / 87
⑦	Sensitivity / precision (critical) in %	100 / 83	8 / 75	72 / 81	48 / 97	1 / 100	14 / 91	55 / 83
⑦	Sensitivity / precision (warning) in %	100 / 76	10 / 100	18 / 71	32 / 45	1 / 100	12 / 88	77 / 88
⑧	Time wasted on false positives (minutes)	373	20	54	269	0	25	88
⑨	Analysis time for all reports (minutes)	1083	80	372	735	11	97	510
⑨	Analysis time per source line of code (min.)	0.45	0.03	0.16	0.31	0.00	0.04	0.21
⑨	Min/max analysis time per report (minutes)	0 / 61	0 / 25	0 / 25	0 / 61	5 / 6	0 / 18	0 / 25
Ⓐ	Avg. time to find a warning true positive	5.13	4.21	4.96	9.42	5.50	3.88	3.57
Ⓐ	Avg. time to find a critical true positive	15.25	13.33	7.29	21.62	11.00	9.70	13.08
Ⓑ	Similarity (Jaccard) to optimal profile	1.00	0.09	0.34	0.31	0.01	0.12	0.64
Ⓑ	Similarity (Jaccard) to opt. critical profile	1.00	0.08	0.61	0.47	0.01	0.14	0.49
Ⓒ	Avg. critical true positive when run as 2^{nd}	16.0	3.2	38.4	23.2	0.6	5.4	25.4
Ⓒ	... when run as 3^{rd}	11.2	1.9	30.0	16.1	0.3	3.1	15.9
Ⓒ	... when run as 4^{th}	8.2	1.4	24.7	11.7	0.1	2.2	9.4
Ⓓ	Avg. additional total effort when run as 2^{nd}	226.1	42.8	245.6	614.8	8.6	59.0	385.8
Ⓓ	... when run as 3^{rd}	177.5	23.9	161.8	534.1	6.8	37.9	300.6
Ⓓ	... when run as 4^{th}	147.9	15.7	110.9	483.6	5.6	26.7	245.2
Ⓔ	Avg. add. effort per true positive when 2^{nd}	14.5	13.37	6.40	26.50	14.33	10.93	15.19
Ⓔ	... when run as 3^{rd}	16.5	12.58	5.39	33.17	22.67	12.23	18.91
Ⓔ	... when run as 4^{th}	19.1	11.21	4.49	41.33	56.00	12.14	26.09
Ⓕ	Predict functions with many warnings, R^2	0.43	0.02	0.00	0.12	0.01	0.21	0.24
Ⓕ	Predict functions with many criticals, R^2	0.40	0.00	0.19	0.12	0.00	0.01	0.08
Ⓖ	Perceived usability	-	++	0	0	+	+++	+

is good on critical faults but performs even better for warning faults. Data in the Ⓒ rows presents the number of critical faults the tool would find when run as second, third or fourth tool after the other tools. The value given is the average of permutations of other tools run before it. The added benefit of running an additional tool decreases with each tool, and follows a tangential curve towards the unique contribution (⑤). The additional effort required for analysis of reports when running the tool after other tools Ⓓ decreases as well: True positives detected by the earlier-run tools need not be analyzed again because we assume that they were fixed. We limited the cost of false positives that were already detected by earlier-run tools to no more than five minutes each, because we assume that a developer will still possess knowledge from analyzing the report earlier that helps him judge the report as false positive quicker. With regards to the average additional effort per critical true positive Ⓔ, we see no clear trend: for ToolC and ToolF, the effort per true positive ratio increases, for ToolB, it decreases, and for the others it remains more or less the same. R^2 values of Pearson correlation strengths (coefficient r squared) given in Ⓕ rows represent how well fault-prone functions can be predicted using the raw number of reports (true and false positives, including consecutive faults) issued for them; i.e., how much of the variance in fault-proneness of functions is explained by the number of non-judged reports issued by the tool. The two Overall figures use multiple linear regression adjusted R^2_{adj} values to predict fault-prone functions from all tools' non-judged reports. Perceived usability Ⓖ is a rough assessment of how well usability of the tool was perceived by our verification experts. Factored in are efforts for learning, runtime (waiting for results), a nice interface but also machine-readable output to allow for integration in custom development environments.

TABLE III
PEARSON SIMILARITY R FOR TOOLS' DETECTION PROFILES

	ToolA	ToolB	ToolC	ToolD	ToolE	ToolF
ToolA	1.00	0.18	0.14	-0.03	0.40	-0.07
ToolB	0.18	1.00	0.23	0.03	0.31	-0.21
ToolC	0.14	0.23	1.00	0.03	0.30	-0.31
ToolD	-0.03	0.03	0.03	1.00	-0.04	-0.14
ToolE	0.40	0.31	0.30	-0.04	1.00	-0.16
ToolF	-0.07	-0.21	-0.31	-0.14	-0.16	1.00

b) Correlation of tools' reporting profiles: A tool's detection profile is a vector that has entries of 1 for each fault correctly detected and reported by the tool, and 0 for each fault that it missed. False positive and true negative reports are left out. For these vectors, the Pearson correlation coefficient r is a measure of similarity. Table III lists coefficients of detection profile similarity between pairs of tools. Values closer to 1 resemble high similarity, while values closer to -1 denote that the tools complement each other.

IV. Discussion

This section gives answers to the research questions, and discusses threats to validity.

A. Is it justified to apply ASV to already qualified software?

The question has two perspectives: an absolute quality improvement perspective and an economic view.

a) Quality improvement perspective: Wagner et al. [36] compare static analysis reports to bug reports/fixes, and find a rather poor connection. Ayewah et al. [6] summarize that only 4% of faults are detected, and attribute that to the analysis method (pattern analysis). We evaluated tools like ToolA or ToolD that indeed were not very sensitive to critical faults. Yet other tools are almost ten times more sensitive to critical faults. ASV tools issued a total of 86 critical and 184 warning reports. Out of these, 70 critical (83%) and almost 140 warning (76%) reports were true positive, i.e., indeed faulty code. So, the answer to the first view is: yes, applying ASV is justified because it will still detect faults.

b) Economic perspective: ASV has been found to be economic compared to later tests and validation if applied in time [11], [40]. In our case, however, SuT was already verified and validated. Still, the average time to find a true positive is ≈ 5 minutes (warning) and ≈ 15 minutes (critical). Whether this effort is too much depends on one's quality goals, and schedule reserve. As a ballpark figure, one should calculate with a 10 to 30 seconds analysis per source line of code plus effort for fixing plus set up costs. For a 40,000 LOC software this easily sums up to several months of developer effort.

B. Are there significant differences between different ASV tools' capabilities?

In their seminal work, Young and Taylor [39] postulate that no single practicable fault-detection technique is capable of finding all faults. Based on theory, we expect to find differences. Two groups — three if we put ToolD on its own — can easily be discerned. ToolB, ToolC and ToolF focus more on critical issues. Context is more important here, they have longer runtimes, more unique critical contributions, are more sensitive to critical faults than the other tools, and cause more effort for judging their reports, also resulting in longer analysis times in total, per line and per true positive.

But tools also differ within their groups. ToolF, for example, is highly sensitive to warning-level faults. The detection profiles of ToolB and ToolF complement each other in several aspects. In general, pairwise similarity of detection profiles is barely $r > 0.3$, and only one $r \geq 0.4$. Almost every tool has several unique contributions to detection of critical faults.

There are significant differences between ASV tools' capabilities. It may be worthwhile to combine tools (like ToolF) with other tools because detection profiles are so different.

C. What is the best ASV tool available?

Sadly, we are unable to answer this question. Firstly, we only surveyed six tools. While we are confident to have analyzed top-notch tools, there may be better ones among the dozens of other tools. Secondly, there are so many parameters that influence evaluation — like grouping/consecutive faults, context, warning vs. critical, handling of non-fault defects, iterative use, usability, learning curve, developer effort, license costs[2], and runtime; some of which we have not even addressed in our study — that it is impossible to state a universally valid answer.

Third, as far as the space domain with its extreme failure costs is concerned, a good choice may be tools with high sensitivity to critical faults and maximum unique contribution. So ToolB may be the first choice. But using a second or even third tool may be considered. ToolC heavily contributes to critical fault detection when run second or third, while ToolF is rather complementary to ToolB and is strong on warning-level faults. ToolE scores by wasting few time on false positives and low time to find a true positive, high precision, fast runtime, and nice usability features. ToolA has, for example, very low analysis times per line of code and per true positive warning, and very high precision regarding warning messages.

D. Does longer analysis runtime mean more precise reports or better results?

Practitioners evaluating tools might be tempted to think that longer runtime means more sophisticated analysis, or, for usability reasons, favor short-running tools. However, there are not many conclusions that can be drawn from tool runtime alone. There seems to be a relation between longer runtime and a focus on critical faults; yet ToolF is an exception to this rule. Likewise, longer analysis time seems to be associated with more complete detection (sensitivity) of critical faults; with ToolF again being the exception. For warning-level faults, there is no indication of such a relation. Regarding precision, there seems to be no correlation with runtime.

Longer runtime does not imply higher or lower precision. While there seems to be some relation from longer runtime to the quality of results (higher sensitivity to critical faults, focus on critical-level report), there are exceptions to this.

E. Are tools that issue more reports less cost-efficient?

A tool that issues more reports causes more effort for checking these reports. Software developers (or their managers) might therefore be reluctant to use such a tool, and instead, use tools that issue fewer reports (cf. [11]).

Indeed, the analysis cost per line of code is higher for tools that issue more reports: ToolB, ToolC and ToolF issued over 100 reports each, and cause analysis efforts per line of code greater 0.15 minutes, while ToolA, ToolD and ToolE cause efforts < 0.05 minutes. The three tools also cause more time wasted for false positives. But this trend is not observed for average time to find a warning true positive. All tools except for ToolC cause similar efforts between three and five minutes per warning-level fault. For critical faults, ToolA, ToolC, and ToolF cause higher efforts but the difference is rather small.

[2]A single tool may cost up to several ten-thousand EUR per year, which may a be relevant figure even in a business environment.

To sum the answer up: Tools that issue more reports are more costly to use but they are not less cost-efficient. Cost-efficiency is not a reason to not apply tools that issue more reports. Reducing cost is a reason to not apply such tools, but at the risk that undetected faults cause problems and consume efforts later on. In particular, even the large amounts of orange reports from tools like Code Prover (see Section II-B2a) may be worthwhile being dealt with.

F. Is it effective and efficient to apply more than one tool?

We address both parts of the question separately.

1) Is it effective to apply more than one tool?: A precondition for improving quality is that applying different tools leads to more true positive findings being made. Only then can more faults be fixed, and thereby quality be improved. This section investigates the question whether using more tools adds valuable information about existing faults.

"Effective" means that applying another tool adds benefit to software quality. Emanuelsson and Nilsson find that static analysis tools "provide largely non-overlapping functionality" [11]. With our broader definition of automated software verification, we obtain even more diversity. It suggests itself that additional tools will find more true positives, improve quality, and hence be effective. Our data supports this: All tools (except for ToolD) find true positives even when run as second, third, fourth, or even as sixth tool due to the observed unique contribution by every tool.

The raw number of reports generated by each tool can explain only little of the variance in fault-proneness of functions (Figure II, rows Ⓕ). The maximum for a single tool is 0.24 for warnings and 0.19 for critical faults. Combining several reports, $R^2_{\text{adj}} = 0.40$ (critical) and $R^2_{\text{adj}} = 0.43$ (warning) of the variance is explained. This could possibly be exploited to cheaply identify fault-prone functions by merely counting reports of several ASV tools.

2) Is it efficient to apply more than one tool?: We can approach efficiency from a short- and a long-term perspective.

a) Short-term view: Efficiency compared to reviewing: Reviews are known to be economically efficient. Hence, we can decide efficiency through comparison with reviews. Zheng et al. [40] claim a $30-40\%$ efficiency benefit of FlexeLint and Klocwork. Klocwork was part of our study but we know nothing about FlexeLint. So we first estimate bounds for the efficiency of FlexeLint.

On the one side, we know from [40] that it generates twice as many reports as Klocwork, giving us an approximate lower bound. On the other side, one study [11] discarded Code Prover because it generated more reports than FlexeLint. So effort for FlexeLint is somewhere between Klocwork and Code Prover, which are both part of our study. Furthermore, FlexeLint reports are easier to analyze because they are more shallow [11]. Using our data, we roughly estimate the effort for using FlexeLint with 15 minutes per true positive, which is $30-40\%$ [40] below review effort. Klocwork, as a lower bound, is, of course, in the same range. Hence, an effort of less than 15 minutes plus $30-40\% \approx 20$ minutes per critical true positive is efficient. Figure 1 shows analysis effort compared to critical faults found. Most tool combinations are above the review efficiency line, implying more true positives in shorter time; and the ones below it are not far off.

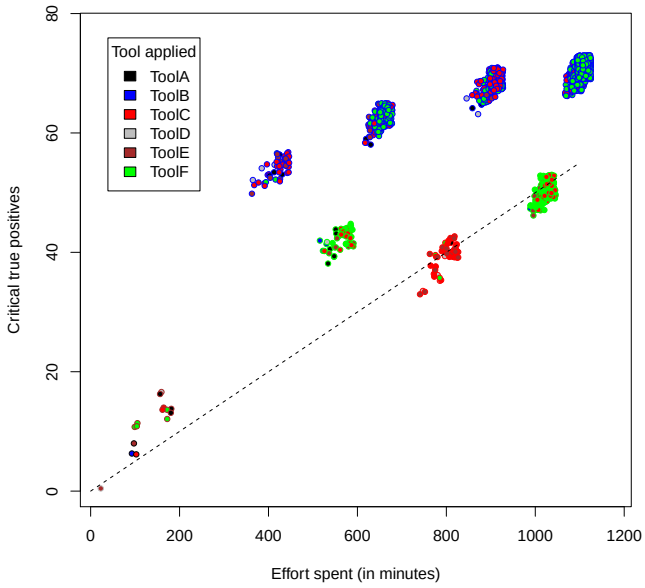

Fig. 1. Analysis effort vs. critical faults detected by tool combinations. Marks colored according to main contributor; center colored with second main contributor. Marks shifted by small random offsets to emphasize visible effect. Dotted black line is the review efficiency.

b) Long-term view: Customer perspective: The cost of quality consists of positive and negative cost. Positive costs are spent for achieving higher quality like testing and static analysis. Negative (or non-conformance) costs are costs incurred because the software is lacking in quality, e.g., causing a software failure leading to loss of a spacecraft, repair costs, bad publicity, and so on (cf. [31]). The sweet spot of quality is where the sum of both costs is minimized.

Let us assume that a satellite costs about 200 million EUR. 10% of development and production efforts are spent on software, so risks originating from software might roughly be 10% as well. If an approximate 20% of science missions fail, we estimate a $10\% \times 20\% \times 200 = 4$ million EUR software risk in a mission. Reducing failure risk by 10% saves about 400K EUR [14], [31]. Soley [34] estimates that basic code faults account for 10% of faults in production code. If these 10% could be obtained through ASV, it was efficient to invest up to 400K EUR in ASV. Given a cost of 400 EUR per line of flight software code [4], we have 50,000 lines of code. ASV with 0.45 minutes average analysis time per line of code for analysis with all tools (Table II) at 2 EUR per developer minute, is far below 400K EUR and leaves a margin for things like license costs, or configuration efforts.

G. Would a simpler evaluation method lead to comparable results?

We invested considerable efforts in the tool evaluation. When we had our debriefing discussions with the vendors, they told us that our evaluation was probably the most sophisticated analysis done so far. Often interns and students are tasked with doing rather superficial tool evaluations. But the necessary expertise, and the efforts for configuring (and running) tools and consolidating and judging reports should not be underestimated. Practitioners will perhaps have to pick tools more by gut feeling than informed decision because of the effort otherwise necessary.

Would we have come to similar results with a simpler approach? For several reasons, we do not think so. If the evaluation is to be fair, it cannot be very much simplified.

For example, a simpler evaluation method might merely look at the number of reports. However, the R^2_{adj} values of the individual tools show that the number of reports can barely explain the variance in fault-proneness of functions. Such methods must therefore be ruled out.

a) Relevance of context: Applying tools to code that has been specifically crafted to a certain fault is like applying them in a without-context view. Instead of crafted examples, we used real-world code. This code is embedded in a larger application and therefore has a call context. The true strength of many ASV tools will only show in the presence of context.

For tools like ToolB, ToolC and ToolF, true or false positive decisions changed up to 20%; whereas other tools were not affected. Critical reports in the without-context case may become false positives in the with-context case, but there are no critical reports that become false positives from with- to without-context case (16 vs. 0 reports). For warning reports, it is the other way around: More without-context warnings become with-context false positives than with-context warnings becoming without-context false positives (4 vs. 14 reports). To summarize: Context reduces the amounts of critical and certain kinds of warning reports, but increases warnings of other types, e.g., ones associated with defensive programming.

b) Clocking analysis time: For obtaining meaningful times for judging reports, the code analyzed must have a real-world context. And, of course, the experts doing the judging should be knowledgeable of the domain as well as software verification. We could answer questions regarding the efficiency of ASV on the basis of clocked analysis times.

c) Consecutive faults: Handling of consecutive faults poses a severe problem for consolidation of the report list. Determining whether a fault is merely a consecutive fault is a considerable additional analysis effort. However, this information is needed, otherwise the results of two tools (one reporting and one not reporting consecutive faults) cannot be compared. A study that ignores the issue of consecutive faults will be extremely biased against or in favor of tools like ToolE.

d) Lessons for future evaluations: Some tool vendors effectively sabotage benchmarking; either consciously through DeWitt-clauses, or (inadvertently) by technical means like not exporting report locations (line numbers). Instead, standardization of report data should be achieved. Open interfaces are needed to make benchmarking easier, to allow meta-tools to incorporate results from other tools, and to provide enhanced verification services. Far more than one thousand hours of work were carried out by experienced industry personnel. We could automate much less of the work than originally intended. If no solution is found for the issues of automating analysis, neither will full-scale in-depth studies be possible, nor can smart meta-tools be provided.

Evaluation of tools allows advances regarding qualities like sensitive or precision in the first place. More and repeated efforts with the help of the research community are probably necessary to aid practitioners with their choice (cf. [8]). The research community should pay more attention to evaluating tools; and to developing evaluation methods. We do not yet have a method for choosing the most suitable tool or combination of tools in a given situation, but this is what practitioners need.

Future work should include more tools and more diverse code. It should address creating an open and evolving database about code, and true and false reports. Tools should be applied iteratively, fixing bugs in between. However, this would further complicate data collection. Usability and psychological aspects need to be addressed as well. For example, long runtimes reduce interactivity, require discipline, and may be less rewarding for developers (cf. [11], [38]). Getting dozens of such reports might be perceived as much more frustrating than getting shallow and easily judged try-and-fix ones. Long runtimes might need to be reflected in project schedules, because fixing must start long before the end of the project to accommodate the long waiting times in the process. Also, detection of non-fault defects and their immediate and long-term effects should be regarded (e.g., technical debt) in evaluations.

Last but not least: While in-depth evaluation of ASV tools is costly, one learns a lot about the tools and their methods.

H. Limitations and Threats to Validity

We think we found fair compromises for dealing with the issues of context and consecutive faults. Yet both aspects can influence evaluation results.

Our 10%-sample (60 functions) probably generalizes to the whole project. But generalization to other projects may be limited. SuT has its peculiarities like being a space flight middleware, being qualified at time of analysis, having 90% unit test coverage, and passed one analysis with ToolB after half of the duration (and was probably frequently checked with ToolD.) One should therefore not assume that one tool is, in all situations, better than another. Instead, a method is needed to come to a well-grounded decision about which tools are more suitable in a given situation.

We did not modify the SuT code in response to confirmed reports, and then re-run tools. Tools only had a single shot (after the configuration optimization). And we do not know about faults that were not detected by any tool. Report analysis times were clocked only once per report. There are

probably variations between experts; although they all were experienced. The fault catalog tried to standardize how to judge individual faults but such decisions may depend as well.

V. Related work

Psychological aspects of static analysis are studied by Ostberg et al. [26]. Ayewah et al. [5] analyzed the use of static analysis tools in the broader scope of organizational policies. They conclude that tools are an important asset for finding faults before production. Similar benefits are seen in academic environments where automated analysis tools can provide quick, automated and objective assessment of code quality, and thereby improve quality and readability [23]. Hovemeyer and Pugh [17] stress the difference between style checkers and fault detecting tools, and note that the latter can find considerable amounts of faults even in software written by experts. Plösch et al. [29] compare static analysis tool results to human reviews in order to determine how good automated analysis is. They also find that static analysis is valuable for guiding expert reviews [28].

Wagner et al. [36] applied the static analysis tools PMD and FindBugs on software from two projects in order to characterize reports made by the tools. Lessons learned for a corporate environment is presented by Emanuelsson and Nielsson [11]. The experience reports provide interesting qualitative information on the use of such tools. A study by Temmerman et al. [35] is quite detailed in its analysis but it is restricted to MISRA rule checking, not functionality. Ourghanlian [27] presents a case study of Polyspace and Frama-C for nuclear power plants but argues that the number of Polyspace's orange reports was too high to handle them. The domain is similar to spaceflight because power plants are also one-of-kind devices, which one does not want to fail. Shiraishi et al. [32], [33] evaluate several tools with crafted code snippets. Information on creating a test corpus can also be found in [9], [20]. In particular, the NIST Reports on the Static Analysis Tool Expositions (e.g., [25]) provide comprehensive overviews.

Hellström [15] presents a host of different commercial and open source analysis tools. He selects some for further evaluation, focusing on usability, providing false positives and negatives for a simple set of crafted faults. He finds that open source tools are not capable of keeping up with the quality of commercial ones.

VI. Conclusion

Verification of software code is an important component of validation and verification activities to ensure correct software. This is of particular interest in domains and businesses, where a software failure, caused by a fault, can have severe consequences. In our domain — spaceflight — a single failure can mean the loss of a several hundred million EUR spacecraft. To support code verification activities, automated software verification tools analyze source code, in order to identify code that is suspected to not perform as expected.

Although ASV tools have long reached a mature state, and should be a common part of any modern development environment, they still face rejection. Typical criticism is that they find too few real faults, or that they report too many false positives and cause more cost then they bring benefit. For large purchasers of spacecraft with their embedded software, transparency of verification tools' capabilities is a precondition for effective quality assurance. Commercial companies and agencies alike are in dire need; yet reliable and in-depth studies are missing. Therefore a study of six ASV tools to evaluate them with real-world flight software was contracted. We investigated several questions and found that

- ASV tools can find faults efficiently even in software that has been qualified, disarming criticism that they would not find real faults or be too expensive;
- there are big differences between tools and capabilities;
- there is, however, no simple, universally valid answer to the question, what the best ASV tool is;
- longer analysis runtime does not lead to higher precision of results but indicates a focus on critical rather than warning-level faults;
- tools that issue more reports cause higher analysis cost trivially because they report more faults but they are not less cost-efficient;
- it is effective and often efficient to apply more than one ASV tool to find further faults; and
- there are studies that evaluate ASV tools but they often do not go into details enough.

The discussion about consecutive faults emphasizes the advantages of using ASV early, i.e., integrated into development. As further contributions we

- address critical embedded software in a domain that requires it to be dependable and safe,
- report an elaborate study (including clocked analysis times) of several ASV tools,
- include several state of the art tools, using high quality (90% test coverage and validated), real-world software,
- analyze the added benefit from combining tools both from effectiveness and efficiency views,
- propose a structure for a fault database to collect information about true and false positive reports,
- present and discuss a method for evaluating tools, and
- inform practitioners and researchers of several pitfalls that conducting an ASV tool evaluation can have.

We as practitioners want to motivate other researchers to get involved in evaluating tools. There is a need for more transparency in automated software verification, and for well-grounded methods for selecting tools that suit a given situation.

Acknowledgments

The authors thank the verification experts Anton Fischer and Mário Pinto for their contributions to data collection, and Georg Witos for his advice on German law. Data collection has been funded under BMWi contract number 50PS1502 (Evaluierung von Software-Verifikationsmethoden und -Werkzeugen).

REFERENCES

[1] ECSS-E-ST-40C: Space engineering – software, 2009.

[2] ECSS-Q-HB-80-03A: Space product assurance – software dependability and safety, 2012.

[3] ECSS-S-ST-00-01C: Ecss system – description, implementation and general requirements, 2012.

[4] H. Apgar. *Space Mission Engineering: The New SMAD (Space Technology Library, Vol. 28)*, chapter Cost Estimating, pages 289–324. Space Technology Library. Microcosm Press, 2011.

[5] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 1–8, New York, NY, USA, 2007. ACM.

[7] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[8] P. E. Black, L. Badger, B. Guttman, and E. Fong. Nistir 8151: Dramatically reducing software vulnerabilities — report to the white house office of science and technology policy. Technical report, 2016.

[9] P. E. Black, M. Kass, M. Koo, and E. Fong. Source code security analysis tool functional specification version 1.1. Technical report, 2011.

[10] G. Brat and R. Klemm. Static analysis of the mars exploration rover flight software. In *Intl. Space Mission Challenges for Inf. Technology*, pages 321–326, 2003.

[11] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008.

[12] R. Gerlich and R. Gerlich. ESVW-TN-FTC-001 - fault type catalog – catalog of fault types in space software from the esvw investigation, revision 1. Technical report, 2016.

[13] R. Gerlich, R. Gerlich, A. Fischer, M. Pinto, and C. R. Prause. Early results from characterizing verification tools through coding error candidates reported in space flight software. In *Data Systems in Aerospace*, DASIA. Eurospace, 2016.

[14] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and S. Zasukha. A study of orbital carrier rocket and spacecraft failures: 2000-2009. *An International Journal of Information & Security*, 28, 2012.

[15] P. Hellström. Tools for static code analysis: A survey. Master's thesis, Linköpings Universitet, February 2009.

[16] G. J. Holzmann. Mars code. *Comm. of the ACM*, 57(2):64–73, February 2014.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th OOPSLA*. ACM, 2004.

[18] C. W. Johnson. The natural history of bugs: Using formal methods to analyse software related failures in space missions. *LNCS*, 3582:9–25, 2005.

[19] G. Klass and E. Burger. s^2erc project: Vendor truth serum. Technical report, Georgetown University, 2016.

[20] K. Kratkiewicz and R. Lippmann. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[21] N. G. Leveson. Role of software in spacecraft accidents. *Journal of Spacecraft and Rockets*, 41(4), July–August 2004.

[22] J.-J. Lévy. Using static code analysis to hunt down bugs. *INédit*, 60, July 2007.

[23] S. Loveland. Using open source tools to prevent write-only code. In 6^{th} *Intl. Conf. on Information Technology: New Generations*. IEEE CS, 2009.

[24] G. McCall. *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. Motor Industry Research Association, 2004.

[25] V. Okun, A. Delaitre, and P. E. Black. Report on the static analysis tool exposition (sate) iv. Technical report, 2013.

[26] J.-P. Ostberg, S. Wagner, and E. Weilemann. Does personality influence the usage of static analysis tools? an explorative experiment. In *9th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE, pages 75–81. ACM, 2016.

[27] A. Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, pages 212–218, 2015.

[28] R. Plösch, H. Gruber, G. Pomberger, M. Saft, and S. Schiffer. Tool support for expert-centred code assessments. In *Proceedings of* 1^{st} *International Conference on Software Testing, Verification, and Validation*, ICST, pages 258–267, april 2008.

[29] R. Plösch, A. Mayr, G. Pomberger, and M. Saft. An approach for a method and a tool supporting the evaluation of the quality of static code analysis tools. In *SQMB Workshop, held in conjunction with SE 2009 conference*, 2009.

[30] C. R. Prause, M. Bibus, C. Dietrich, and W. Jobi. Tailoring process requirements for software product assurance. In *Proceedings of the International Conference on Software and System Process*, ICSSP, pages 67–71. ACM, 2015.

[31] C. R. Prause, M. Bibus, C. Dietrich, and W. Jobi. Software product assurance at the german space agency. *Journal of Software: Evolution and Process*, 28(9):744–761, 2016.

[32] S. Shiraishi, V. Mohan, and H. Marimuthu. Quantitative evaluation of static analysis tools. In *IEEE International Symposium on Software Reliability Engineering Workshops*, ISSREW, pages 96–99, Nov 2014.

[33] S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In *IEEE Intl. Symp. on Software Reliability Engineering Workshops*, ISSREW, pages 12–15, Nov 2015.

[34] R. M. Soley. How to deliver resilient, secure, efficient, and easily changed it systems in line with cisq recommendations. Technical report.

[35] M. Temmerman, H. V. Hove, and K. Bellemans. Kricode research report 1: Comparative study of misra-c compliancy checking tools. Technical report, TERA-Labs, 2012.

[36] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *1st International Conference on Software Testing, Verification, and Validation*, pages 248–257, April 2008.

[37] A. Witze. Software error doomed japanese hitomi spacecraft. *Nature*, 533:18,19, May 2016.

[38] S. Wray. How pair programming really works. *IEEE Software*, 27:50–55, 2009.

[39] M. Young and R. N. Taylor. Rethinking the taxonomy of fault detection techniques. In *11th International Conference on Software Engineering*, ICSE, pages 53–62, New York, NY, USA, 1989. ACM.

[40] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006.