embeddedworld2023
Exhibition&Conference
...it's a smarter world

Reconciling Software Development Speed and Robustness with Optimally Balanced Static Application Security Testing

Jacob Palczynski
Training Services
The MathWorks GmbH
Aachen, Germany
jpalczyn@mathworks.com

Martin Becker
Application Engineering
The MathWorks GmbH
Munich, Germany
mbecker@mathworks.com

***Abstract*—Developing and maintaining secure software is essential for modern safety-critical systems but is also a challenging problem for agile development teams. Proven security and fast development are natural antagonists which must be reconciled to minimize vulnerabilities while guaranteeing a fast response to cyber incidents. In this paper, we highlight the most common problems in this context and show a proven strategy that makes "agile security" an achievable routine. The proposed strategy is based on the best practices of industry leaders from various application domains and their use of static code analysis at the right time and with the right scope and depth. It addresses the well-known resource problem (who does security and when?), the learning problem (how do developers learn, how do teams improve?), and how to deliver sufficient and consistent security evidence. With a clever balance of tools, automation, and feedback, cybersecurity can be quantified, incrementally improved, and delivered on time.**

Keywords—DevSecOps, cybersecurity, agile, formal verification, static code analysis, waste, efficiency

I. INTRODUCTION

Cybersecurity is a big challenge for today's software, with more and more attacks aiming at embedded systems. For example, the number of cyber incidents in automotive systems has more than doubled from 2021 to 2022, with 89% of attacks involving embedded systems [1]. The goal of cybersecurity is to protect the system from its environment, to resist such attacks and prevent malicious access. This contrasts with functional safety, which vice versa seeks to protect the environment from system malfunctions. Although it seems that cybersecurity is the less critical among these two, it is important to understand that unprotected access can lead to undesired control over the system and impair functional safety. Therefore, cybersecurity is not only an add-on, but the foundation for functional safety.

Attacks on embedded systems are not fundamentally new, but until recently were not explicitly addressed in most industry standards. Traditional safety analysis, founded on probabilities and statistics (e.g., Fault Tree Analysis), falls short for cybersecurity. Attackers often actively search for security issues with massive expenditure of resources. Therefore, it is not the probability that counts for security, but the feasibility. Especially safety-critical software must function correctly even in unlikely situations, which means that malicious attacks must be explicitly considered during system design. Several newly published standards, such as ISO/SAE 21434 , IEC-62443 [2], and DO-356 [3], fill this gap now with new security requirements.

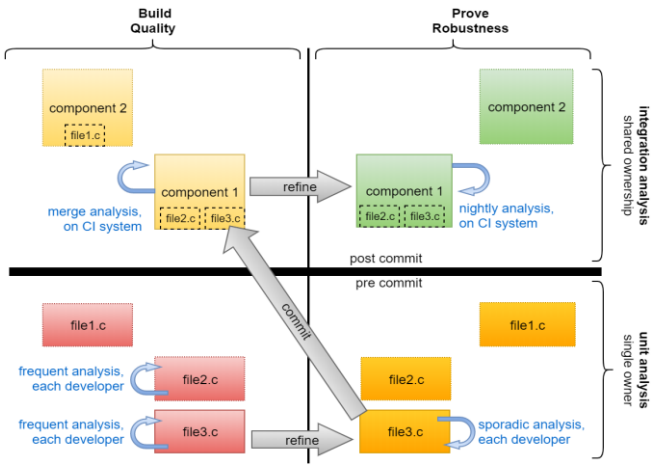

Figure 1: Optimal Workflow: Quality first, robustness second - local first, global second. Most iterations take place early and on small units.

Developers, integrators, product managers, and other contributors must familiarise themselves with this new topic and follow these latest standards and regulations. However, only few cybersecurity experts are available in the job market, and most do not have the technical background of the respective application. Apart from a reliable process, a way must be found to transfer this knowledge to existing domain experts. Furthermore, any security vulnerability discovered must be closed quickly to limit financial, operational, safety, and privacy damage. In summary, the development process must not “only” satisfy new security requirements, but must also be more agile than before.

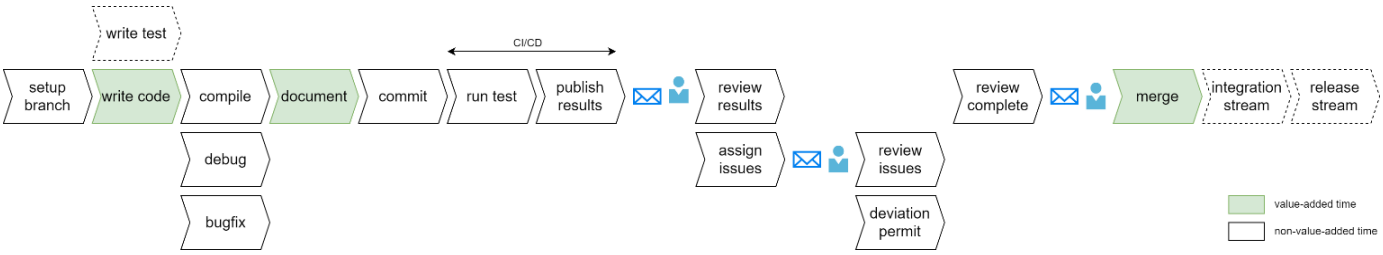

Figure 2: Exemplary software development process. The amount of non-value-added time ("muda") is usually dominant.

In this paper, we show how an optimal development process for cyber-secure software can meet these new requirements and simultaneously enable significant quality improvements. We focus on the efficient assurance of quality using Static Code Analysis. Our methods are based on the proven principles of *DevOps* and the *Toyota Production System*. Both aim to establish quality as early as possible (*Shift Left*) while shortening development cycles and reducing costs. Essential elements include avoiding unnecessary work, automation, and individual and organisational learning. More details are provided in the following sections.

The principles shown here lead to higher software quality in less time [4], [5] as observed by the authors and as comprehensively documented in literature [4], [5].

II. New Challenges from Cybersecurity

As explained earlier, cybersecurity cannot be adequately covered by traditional methods and processes as explained earlier. Therefore, we first highlight the new challenges that many development teams are currently facing.

A. Updates Must Be Expected

Software considered secure today may already be insecure tomorrow. One obvious reason is that weaknesses can be missed during development and discovered only during operation. Another reason is that vulnerabilities may stem from external sources. For example, the "Meltdown" vulnerability was caused by a hardware design flaw, but demanded software updates since only hardware platforms could be fixed by firmware updates [6]. Therefore, even with "perfect" software, we must anticipate updates to patch newly identified security vulnerabilities.

These updates may require multiple iterations through the development process (see *Figure 3*), which is why even minor inefficiencies can accumulate exponentially higher costs. A good analogy to this cost increase is the compound interest effect known from mortgage loans – a seemingly small price which is paid frequently, can accumulate to a major penalty over time. An effective process, maintainability, and modularisation of the

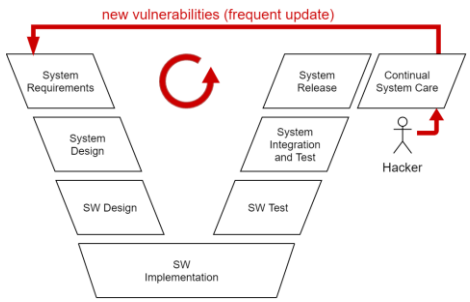

Figure 3: Cybersecurity requires regular and thus repetitive development processes, illustrated here by the example of the V-model.

software are crucial to avoid this. These aspects are part of fundamental software quality and become the basis for maintaining cybersecurity throughout the product lifecycle. As discussed next, programming guidelines offer an approach to achieving this fundamental quality.

B. Compliance with Programming Guidelines Is Insufficient

Programming guidelines such as MISRA C or CERT C [7] and avoiding dangerous code patterns (CWEs) reduce the potential for errors during development and contribute significantly to code quality. However, compliance with these guidelines is only weak evidence for cybersecurity. They cannot guarantee that the software works reliably in all corner cases [8] since they mainly focus on bad design patterns.

(1) **c**onfidentiality (data is kept secret or private),
(2) **i**ntegrity (data is trustworthy and free from tampering),
(3) **a**vailability (software works as and when expected).

These properties are also known as the *CIA Triad* [9], an integral part of the new cybersecurity standards and regulations [10] [3]. Covering the CIA Triad requires additional evidence achieved through additional verification methods.

C. Evidence Requires (Waiting) Time

Obtaining strong security evidence takes time. The more confidence is required, the more verification time must be invested, especially as code complexity increases. Thus, we have the choice between fast yet weak results or more time-consuming but stronger results. The former may be incomplete and may result in several design iterations. On the other hand, the latter may require more initial effort from developers and testers but fewer iterations to fix all programming errors.

Theoretically, development could happen in parallel with verification, separating both efforts to a certain degree. For example, developers could work on the next feature while their latest changes are under verification. However, any software changes may lead to stale verification results. Consequently, development and verification must be at least partially sequential, requiring a clever balance between speed and depth of verification activities.

D. Who Trains the Developers?

In principle, independent from cybersecurity, there is also a resource and learning problem: People make mistakes. Because of the broad nature of cybersecurity, the resulting defects can no longer be identified and resolved by a small QA team. Research shows that most defects are due to individual mistakes and lack of knowledge [11], so it is worth starting here. Every single developer must write secure code from the beginning.

But how can developers identify errors that can lead to security vulnerabilities and learn from them as early as possible and systematically? Possible answers are code review and pair

programming [12], but these do not work as well in cybersecurity. There are not enough security experts to make this a viable approach. Moreover, it is well-known that humans are ill-suited to mentally consider all corner cases, write comprehensive tests, and reliably determine the absence of defects [13]. Therefore, code review and pair programming are typically focused on architectural decisions and functional correctness and are ill-suited to cover cybersecurity.

Finally, developers typically get quality feedback with a time delay, which is a problem. For example, this can happen when integration problems arise or even during the running operation of the product. Not only do developers lack a sense of the completeness of security properties or the effectiveness of hotfixes, but overall costs are disproportionately increased by late discovery [14].

Summary: Cybersecurity requires demonstrably robust and maintainable code in a short time. The development process must be designed so that as few iterations as possible take place and are terminated as soon as sufficient evidence for security properties is obtained. Towards this, developers must be trained, and their work made measurable. Nevertheless, there must be room for flexibility along the way so that new features can be implemented quickly. While cybersecurity must be a part of every process step, it must not slow the process down unnecessarily. These goals coincide with the principles of two well-known production systems, which we will explain in more detail in the following section.

III. What Makes an Efficient Process?

The *Toyota Production System* (TPS [4], also known as *Lean* in Western cultures) describes practices that have been applied successfully for decades, reconciling quality and development speed with maximum resource efficiency. These practices started in vehicle manufacturing (Toyota) but have long reached other disciplines, such as software development. Close relatives are the more recent *DevOps* practices, which are strongly influenced by TPS and continue the success story. The practices focus on company philosophy, culture, and process. However, they do not prescribe specific tools or workflows but build on basic principles which each company must realise in its way.

Many teams already know that such practices help them to achieve their goals. Results are among others [4] [15] [5]:

(1) Shorter development times for new features.
(2) Fewer defects and higher reliability.
(3) Shorter response times for hotfixes.
(4) Significantly higher profits.

In this paper, we focus on TPS because it is more detailed than the more common DevOps and includes the latter at its core. Thus, the strategies described here apply to both worlds.

A. Avoiding Unnecessary Effort (Muda)

The central objective in TPS is avoiding unnecessary effort or waste — *Muda*. Specifically, *Muda* is defined as any activity that does not directly contribute to product value from the customer's point of view. Examples are rework caused by defects, waiting times, the development of components that do not go into the product, and tasks that serve the preparation (setup of tools) or the handover (reviews and approvals).

The share of value-adding efforts in a software development process is only small, as shown in *Figure 2*. Each step is shown only once, but software development is typically iterative, which makes *Muda* even more important. For example, we must repeat the steps to set up a branch (or at least check it out), develop the fix, commit, re-test, and so on if a defect is found. The entire process must be repeated many times before the software is ready, which further exacerbates *Muda*. While this is unsatisfactory, it is also an opportunity that TPS and DevOps are taking on.

B. Unnecessary Effort (Muda) in Software Development

The TPS [4] defines seven types of *muda*, which can be mapped to software development as follows:

(M1) **Overproduction** (e.g., useless features, increasing almost all other types of *muda*).
(M2) **Waiting time (**e.g., for compilation, automatic testing, or computational resources).
(M3) **Transport (**e.g., handover/handoff, or even data transfer, causing time loss).
(M4) **Overprocessing** [e.g., higher complexity than needed, due to bad or needlessly complex design — causes time loss and increases (M7)].
(M5) **Large inventory** (e.g., work-in-progress such as numerous half-finished features or hotfixes, increasing risk of obsolescence, conflicts, and defects).
(M6) **Setup/use workstation** (e.g., switching between branches, which takes both mechanical setup work and time to mentally switch context).
(M7) **Defects** require bug fixes, retesting, etc. and it is well documented that defects found late are disproportionately more expensive to fix [14].

Note that some activities may be considered *Muda* by definition, yet may still be unavoidable (e.g., process reasons, compliance reports). The updates already mentioned are another example. In theory, these are unnecessary because with complete information about the future, attacks could be foreseen from the start and the software developed accordingly. For such "useless" activities, we follow the maxim: *As efficiently as possible, as rarely as necessary.*

Some principles have been defined in TPS and DevOps (see *Table 1*) to systematically minimize the waste described above, which we will discuss later at the appropriate point.

Table 1: The 14 Principles of the Toyota Production System [4].

Aspect	Principles
Philosophy	P1: Take long-term management decisions
Process	P2: Implement a one-piece flow
	P3: Use pull systems
	P4: *Heijunka* – levelled workload
	P5: *Jidoka* – built-in quality
	P6: Standardize work
	P7: Visual management
	P8: Use reliable and proven technology
Human and Partners	P9: Grow leaders who understand
	P10: Grow exceptional people and teams
	P11: Respect network of partners and suppliers
Problem Solving	P12: *Genchi genbutsu* - go see for yourself
	P13: Take decisions slowly by consensus
	P14: *Hansei* and *Kaizen* – reflect and improve

IV. The Ideal Workflow

In this section, we introduce an ideal software development workflow that minimizes useless effort and addresses the aforementioned challenges of learning problem, resource problem, and sufficient evidence for cybersecurity. It also reconciles high software quality with fast development cycles. The focus lies on the implementation and verification of the source code.

We rely on a balanced combination of pre-commit and post-commit code scans using Static Code Analysis tools. The pre-commit analysis should run *fast* and with *limited context*, complemented by an automated CI job (*Table 1*, P2) which runs after the commit and performs a *comprehensive* proof in the *integration context*. This approach is both reasonable and efficient, as we will explain next.

A. Proofs: Sound Static Application Security Testing (SAST).

As discussed in Section II.B, adherence to coding guidelines is generally insufficient to achieve cybersecurity because rare or unexpected usage patterns are usually absent from functional tests, thus inadequately covering the CIA Triad. What is needed here are "negative tests" beyond the specification. Ideally, all possibilities should be covered entirely, providing strong evidence for robustness. Consequently, methods particularly suitable for cybersecurity should be able to: (1) identify rare and non-intuitive program states, (2a) prove the fulfilment of the CIA properties with sufficient evidence, or (2b) conversely show defects by providing concrete examples. Security then becomes measurable. Such methods exist in the form of static code analysis tools (SAST), which are *sound* (i.e., which consider all reachable program states without missing "bad cases"). We have detailed the advantages over other tools that merely check guidelines in [16].

B. Before Commit: Shift Left Using SAST

It is well known that early verification saves time and costs [14]. Security vulnerabilities can be detected and fixed while being introduced (*Shift Left*, P5) if SAST is applied before the commit, reducing useless effort for later repair (M7). Furthermore, transport (M3) is also minimised since the handover for integration (commit, review) has to be repeated less often.

Specifically, secure coding guidelines such as CERT C and the absence of safety-critical code patterns (CWEs) should be checked early, and so should robustness in corner cases (see above). We will detail the correct sequence later. It is important to cover both to meet cyber-security; see also II.B.

The tools which are used should also minimise the developer's effort. They should be well integrated into the development environment and support the automatic setup of the analyses based on the build environment, thereby reducing setup effort and context switching times (M6). They reduce the time required to fix defects (M7) by providing detailed explanations for the identified security vulnerabilities. Finally, SAST tools must have short response times at the developer's end to minimise waiting times (M2).

The Learning Problem: In the context of *Shift Left*, SAST can also significantly facilitate familiarising developers with the topic of cybersecurity. While pair programming and code reviews are insufficient due to human limitations and lack of experience (see above), the "expert in the computer" reliably closes this gap. SAST can provide rigorous and valuable feedback to developers by working consistently, not missing defects due to human limitations, and acting on security knowledge built into the tool. It provides an initial assessment of code quality within seconds by seamlessly integrating into the programming environment and uncovers poor programming patterns as well as potential security vulnerabilities.

Over time, individual developers will learn and deliver more robust software in a shorter time. Developers can recognise their areas of improvement and reduce errors in the first place by applying the tools repeatedly. Developers profit further from increased confidence that their contributions are robust and ready for integration.

C. After Commit: Less Technical Debt Thanks to SAST

To complement early verification, automated SAST analysis should take place after each commit. The goal is to verify all contributions consistently and in the same target environment (no more "works for me") and ensuring minimum quality before or when the code is integrated. This principle is also known as *gatekeeper* or *quality gate* and can be extended to automated approvals and merges to improve "flow" (P2).

In the event of a "bad" commit, gatekeepers can also implement the principle of *Jidoka* (P5), which means "stop, get together and solve". This is achieved with the "Andon Chord" at Toyota, a rope that runs along the production line. If a worker detects a serious problem, she can pull this rope, resulting in a full stop of the production line after a short grace period [4]. All workers can then come together and focus on solving the problem and preventing defects (M7) from being passed downstream, which would result in even higher costs later. It also prevents similar problems from recurring and not being solved (M3), (M6). It is important to follow up on such incidents by identifying and analysing the root cause (P14) and defining suitable best practices (P6) to avoid identical errors in the future. These practices will lead to efficiency gains by gatekeepers in the long run. For our software development, this can mean enabling additional SAST checks before the commit or introducing appropriate coding guidelines.

D. Scope and Ownership

However, *Shift Left* should not be carried too far because this would also lead to *Muda*. Instead, the scope of analysis should always fit the particular process step. In brief: You should only analyse what you are allowed to change.

A typical mistake with *Shift Left* is all-encompassing early verification, for example, workflows in which each developer performs integration-level analyses before committing. First, this approach appears plausible as it allows catching as many defects as possible early, but it causes unnecessary waiting times (M2) because the analysis produces results that the developer cannot address and will thus ignore. Second, issues may be fixed in a suboptimal way, as some findings will be interface problems that usually affect several stakeholders and should therefore be discussed together. Third, even if the cause of the interface issue is obvious, the developer may take measures resulting in superfluous code in the later integration context because changes in the surrounding code may fix it automatically. One example is redundant defensive coding when the input parameters are not yet known but the final implementation only uses input parameters that do not lead to an error. In this case, redundant defensive coding may lead to performance degradation during

runtime. This all results in overprocessing (M4) due to too much *Shift Left*. And finally, comprehensive early analyses may also lead to developers not accepting or even rejecting the analysis tools. If a SAST tool lacks context and flags too many constructs, developers may ignore its warnings ("I know better") or may not use the tool at all ("too noisy"). In both cases, one can expect more defects (M7), and wasted learning potential.

The correct approach here is a clever balance again, as shown in *Figure 1*: Before the commit (the lower half plane in the figure), only the developer's respective working package (e.g., unit, module) should be analysed. This routine shortens analysis time (M2) and keeps developers' focus in their "territory," where they can immediately make corrections or document intentional deviations from guidelines with the rationale still present. A focused analyses SAST tool must not display issues that depend on external data or control flows to avoid overprocessing (M4) due to lack of context.

On the other hand, a complete integration analysis should occur after the commit, detecting the (few) remaining defects, typically only interface and integration issues. Since integration issues affect multiple parties (shared ownership, upper half plane in the figure), the tool should allow for sharing of the analysis results and facilitate discussions on the issues found as reflected in the TPS principle "Decide with Consensus" (P13). Consequently, SAST tools meant for this scenario should offer tracking, commenting, and assigning of issues, and ideally, a connection to external bug tracking tools.

E. Compliance First, Robustness Second

In addition to the scope of the analysis, the sequence of the aspects to be analysed should be optimised. As described above, software should meet basic quality requirements, such as maintainability before it makes sense to dive into the subtleties of the CIA triad. After all, it is easier to recognise and address the corner cases in a well-structured design, which reduces the learning curve (M6). The horizontal dimension in Figure 1 represents this order exactly. Thus, the aspects should be analysed in the following order:

(1) local quality,
(2) local robustness,
(3) quality of integration,
(4) robustness of integration.

A second reason for this order is the advantage of a faster analysis when no proofs are (yet) required; see also II.C. Following this order also avoids unnecessary waiting times (M2). Conducting a robustness analysis would be inefficient if the software still has many defects, which can be discovered with more lightweight tools.

Finally, there is also an evidence-related reason why a robustness analysis only makes sense *after* the code quality is decent: Too early use of sound tools can lead to incomplete analysis results when "undefined behaviour" is still present in the program. Since the program state is undefined at such locations, a sound analysis must stop tracking offending control flows and thus can hide downstream defects (M7). We have explained this in more detail in [16].

F. Granularity and Frequency of Changes

Both DevOps and TPS require small and frequent changes to avoid overproduction (M1) and minimise inventory (M5). In addition to making root cause analysis easier in the case of a failing gatekeeper (e.g., git blame), this also shortens response times between the creation and detection of integration defects, making bug fixes (M7) easier. In particular, integration analysis should happen periodically (e.g., every night) and not only once at the end of the release cycle.

To ensure that the effort required does not grow proportionally with the analysis frequency, SAST tools must be able to track defects across successive analyses and code changes, and offer an incremental review. Specifically, metadata related to defects (e.g., comments or assignments to developers) must be automatically transferred to subsequent integration analyses since otherwise repeated review activities result in waste (M4). Furthermore, an "old/new" logic should also be available, and comparisons between any two analyses should facilitate spotting trends and narrow down defect occurrences in terms of *time*.

V. One Concrete Walkthrough

We now briefly show how the ideal tool chain for cybersecurity can look like using *Polyspace Bug Finder*™ [17] to check basic code quality and *Polyspace Code Prover*™ [18] for comprehensive proof of robustness. An overview is shown in the following figure:

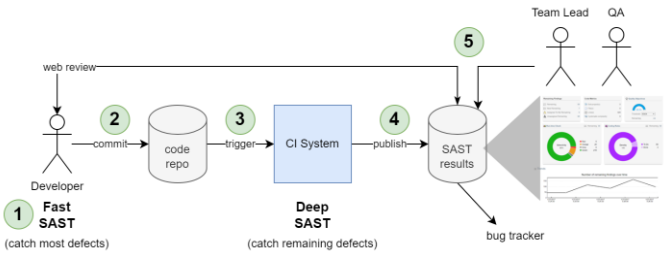

Figure 4: An optimally balanced SAST toolchain.

Pre-commit (1). The code is ideally analysed in early development phases within the developer's IDE (e.g., *Microsoft VS Code*™). *Polyspace as You Code* is used here to provide fast feedback about CERT C violations, CWEs, and other defects. The analysis is limited to individual files specifically, so developers are not slowed down. Setup efforts are minimal since the tool obtains all necessary information (e.g., compiler, include paths, etc.) directly from the IDE. *Figure 5* shows how *Polyspace as You Code* warns the developer about an incompletely initialised variable (`abs_control_output`), which is a CERT C violation. Although the variable is written in the if-else block starting at line 87, the predicates do not cover all possible conditions. The developer can react immediately and prevent a potential vulnerability while working on the algorithm.

Furthermore, integration defects are specifically hidden in the IDE to prevent overprocessing (M4): defects resulting from external input values (e.g., overflows and invalid pointer arguments) could otherwise not be evaluated without considerable effort. The integration context would have to be reproduced as accurately as possible to do this. However, this would only make sense if the recreated context was also verified, which can be solved more efficiently in the next step. If desired, a local robustness analysis using *Polyspace Code Prover*™ can take place at this point as soon as the quality requirements are covered.

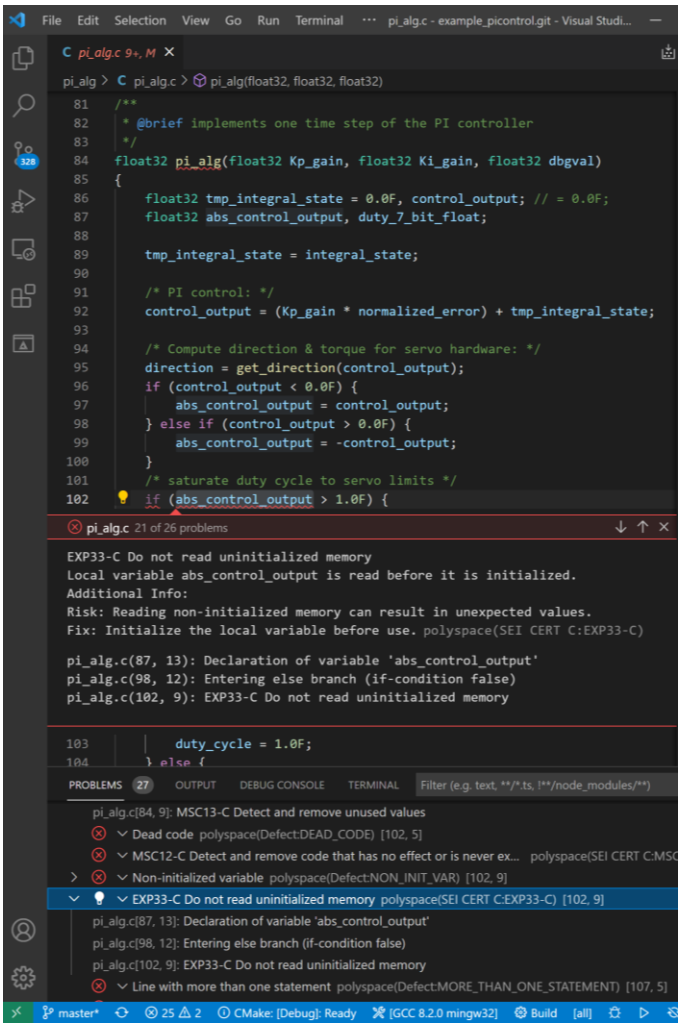

Figure 5: SAST tool "Polyspace as You Code" points out security vulnerabilities early and within seconds in your IDE (here: Microsoft VS Code™. Screenshot used with permission from Microsoft).

Post-commit (3). Integration analysis does not lie in the responsibility of individual developers and therefore is triggered centrally via CI systems, such as *Jenkins*™ or *Zuul*™, performing more comprehensive analyses. The analysis setup takes place automatically as before, this time by hooking into build systems such as *cmake*™, *bazel*™, or similar. Only the desired checks must be selected, which can be done with XML files. With this post-commit step, we can achieve up-to-date, comprehensive, and consistent verification of all code changes.

Both quality and robustness must be checked as before. Robustness analyses in the integration context provide the definitive final evidence but typically require more waiting time than all analyses done up to this point. Therefore, the robustness analysis at the integration level should be performed last but crucially still regularly (e.g., nightly checks) to avoid late and expensive defects.

To limit technical debt (see IV.C), analyses can be automatically evaluated in the CI/CD system, and CI jobs can be conditionally failed with errors when certain criteria are not met (e.g., too many new defects, or violation of certain CERT C rules, too complex functions).

Teamwork and Interactive Review (5). The resulting findings of CI/CD should also be well visualised for efficiency (P7) and allow developers to work together to find a solution. *Figure 6* shows an integration defect in the browser-based centralized review system *Polyspace Access*™ [19]. The review system collects analysis results from the CI runs and provides the entire team with trends, comparisons, and task assignments. Defects are explained through interactive elements in an easily understandable way and do not have to be reproduced again in a time-consuming way. The picture shows a division by zero in `pi_utils.c`, caused by a call from a second file, `pi_alg.c`. It would not make much sense to show this finding to the developer of the first file before the commit since the context would have been missing here, and one could easily have misinterpreted this warning as a *false positive.* On the other hand, the integration analysis provides a concrete example of how the defect occurs and allows the affected developers to meet in the browser interface, agree on possible measures, and open a bug tracker ticket if desired.

If subsequent integration analyses occur, *Polyspace Access* automatically propagates the review information to newer results to avoid costly re-work (M3). Furthermore, developers can show or hide issues found during integration analyses during the early analyses within the IDE, enabling them to focus on their local defects and to see the appropriate context for each issue while bug-fixing and hiding "old acquaintances."

VI. Beyond the Process

Process is not everything. Even the best tools and processes do not protect against all errors (typical limits of SAST tools are discussed in [16]). In addition to the individual programmers, the entire team and the organization must also learn if they really want to exploit the benefits of TPS and DevOps. Therefore, additional measures are necessary for the areas of corporate philosophy, people and partners, and problem-solving behaviour (see the TPS principles in *Table 1*).

Those additional principles are beyond the scope of this paper. However, the following three are worth highlighting:

(1) **Hansei and Kaizen – Look back and continuously improve (P14):** Occurring problems must be analysed down to their root cause (blame-free), and appropriate countermeasures defined for the future. Fixing symptoms is insufficient. In terms of cybersecurity and SAST, this can mean that security vulnerabilities are discussed in a non-judgmental manner in the team; and to check whether software architecture, incorrect use of tools, incomplete solution strategies, or incorrect assumptions may cause the problem to happen again. The team selects one or more problem-solving techniques (e.g., Five Whys, Ishikawa Diagrams, etc.) for this. Suitable solutions and best practices (P6) must also be documented so that improvement can take place.

(2) **Decide with consensus (P13):** Once the problem has been identified, all possible solution strategies should be considered, all necessary data should be collected, and a solution should be agreed upon by the entire team based on the facts. Only then is a solution implemented that can be achieved with team reviews, among other things, as explained in the previous section.

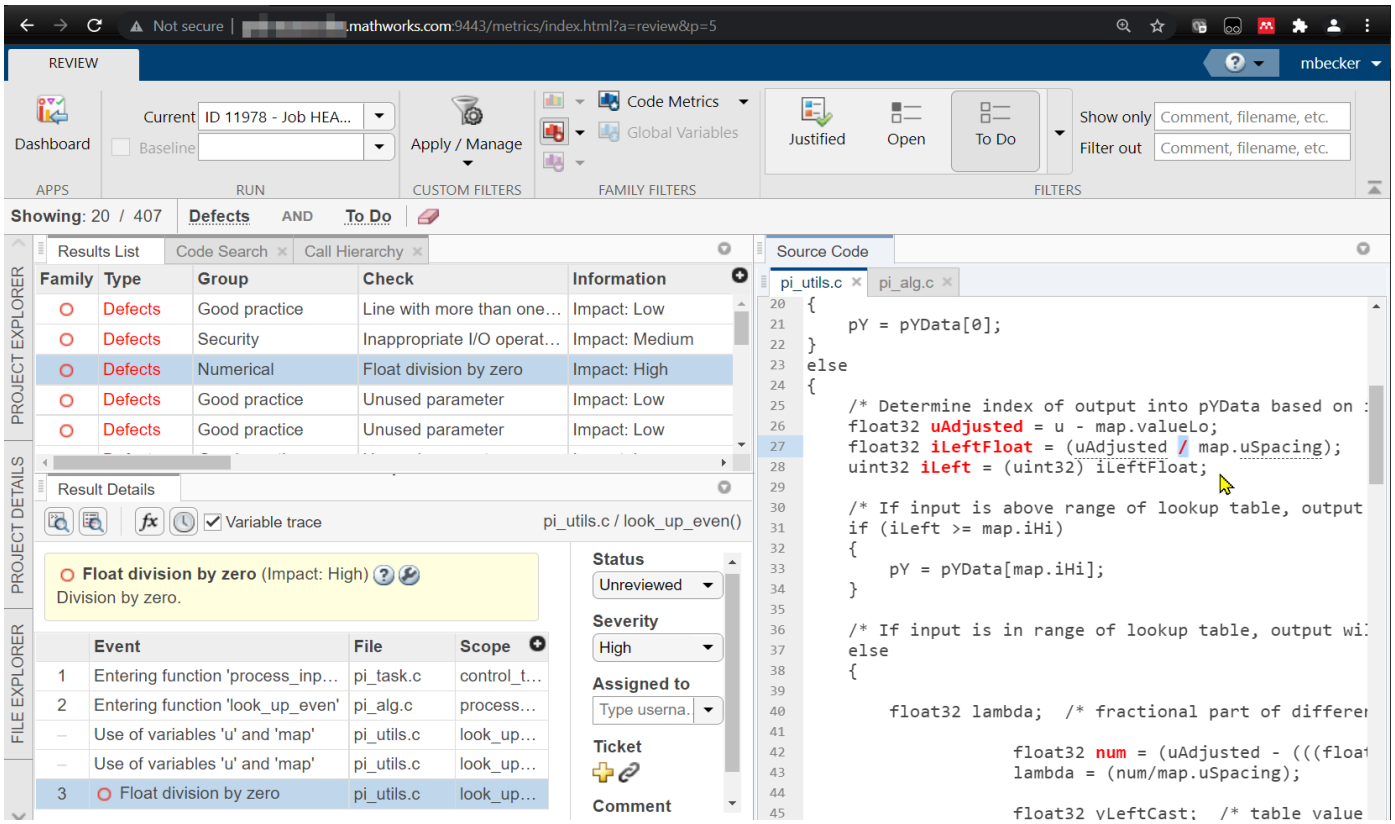

Figure 6: Integration defects are forwarded from the CI/CD system to the browser-based team review, where developers can manage defects and collaborate.

(3) **Develop teams (P10):** Value is created by people, not by processes. We have already elaborated on how individual developers can improve their skills using SAST tools. However, an experienced team is more than the sum of its parts. Together, team members coordinate, learn from each other, and set goals for continuous improvement. In the context discussed here, one option would be successively tightening the gatekeepers so that all developers learn, but no one is overwhelmed. For this purpose, *Polyspace Access* offers freely definable quality levels (*Software Quality Objectives* [20]). These can be increased step-by-step until release, and the team can track how far their software has attained the defined quality and robustness goals in the dashboard.

For further information on the TPS principles and practical examples, we would like to refer interested readers to our primary source [4]. In summary, teams should not only define their process but also make sure they improve it continuously.

VII. Concluding Remarks

Developing secure software in a short time may seem like a contradiction. However, it all depends on the process and culture, as we have shown. A well-balanced workflow can identify vulnerabilities early (*Shift Left*) yet without burdening the developers and testers, and reliably spots programming errors and vulnerabilities. The key is to avoid unnecessary effort ("waste"), which requires selecting the right verification method, the right tools, and the right scope at each point in time.

Static Code Analysis (SAST) tools are an effective way to address cybersecurity challenges in software development. These tools enable teams to identify security vulnerabilities in an automated and consistent way. However, analysing for compliance to CERT C or finding weak patterns is not enough to satisfy the CIA Triad; strong evidence of robustness is required. Such evidence can be provided by *sound* SAST tools and helps closing security gaps that the guidelines leave open. SAST tools can also provide developers with continual feedback on the quality of their work, enabling them to improve their skills over time, deliver more secure software, and minimize unnecessary efforts for additional design iterations.

Finally, the principles described here can and should extend to other design and verification tools to reduce overall waste, and foster individual and organizational learning. Security cannot be achieved with one method only but requires a concerted effort and a deliberately chosen process.

VIII. References

[1] Upstream Security Ltd, "Global Automotive Cybersecurity Report," Upstream, 2022.

[2] International Electrotechnical Commission, *IEC-62443-4, Security for industrial automation and control systems,* 2018.

[3] RTCA, *DO-356, Airworthiness Security Methods and Considerations,* 2018.

[4] J. Liker, The Toyota Way, McGraw-Hill, 2004.

[5] Puppet, State of DevOps Report, Portland, USA, 2021.

[6] Wikipedia, "Meltdown (security vulnerability)," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Meltdown_(security_vulnerability)&oldid=1040587365. [Accessed 1st Oct 2021].

[7] CMU Software Engineering Institute, “SEI CERT C Coding Standard,” [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c. [Accessed 1st Oct 2021].

[8] D. Papp, Z. Ma and L. Buttyan, “Embedded systems security: Threats, vulnerabilities,” in *Conference on Privacy, Security and Trust (PST)*, 2015.

[9] S. Samonas and D. Coss, "The CIA Strikes Back: Redefining Confidentiality, Integrity and Availability in Security," *Journal of Information System Security,* vol. 10, no. 3, pp. 21-45, 2014.

[10] SAE International, *ISO/SAE 21434 Road Vehicles Cybersecurity Engineering,* 2021.

[11] M. Leszak, D. Perry and D. Stoll, “Classification and evaluation of defects in a project retrospective,” *Journal of Systems and Software,* vol. 61, no. 3, pp. 173-187, 2002.

[12] L. Williams, R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the case for pair programming," *IEEE Software,* vol. 17, no. 4, pp. 19-25, 2000.

[13] G. J. Myers, C. Sandler and T. Badgett, The art of software testing., John Wiley & Sons, 2011.

[14] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," NIST, 2002.

[15] C. Fishman, "They Write the Right Stuff," Fast Company, 31 12 1996. [Online]. Available: https://www.fastcompany.com/28121/they-write-right-stuff. [Accessed 02 09 2021].

[16] M. Becker and J. Palczynski, "Increasing Resilience to Cyberattacks through Advanced Use of Static Code Analysis," in *Embedded World Conference*, Nuremberg, 2021.

[17] The MathWorks Inc., "Polyspace Bug Finder," 18 Oct 2021. [Online]. Available: https://www.mathworks.com/products/polyspace-bug-finder.html.

[18] The MathWorks Inc., “Polyspace Code Prover,” 18 Oct 2021. [Online]. Available: https://www.mathworks.com/products/polyspace-code-prover.html.

[19] The MathWorks Inc., "Polyspace Access," 18 Oct 2021. [Online]. Available: https://www.mathworks.com/products/polyspace-code-prover.html#access.

[20] The MathWorks, Inc., "Software Quality Objectives for Source Code," 2010. [Online]. Available: https://www.mathworks.com/discovery/software-quality-objectives.html. [Accessed 20 Oct 2021].

[21] K. Goseva-Popstojanova and A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities,” *Information and Software Technology,* vol. 68, 2015.