

Deploying Static Application Security Testing on a Large Scale

Achim D. Brucker and Uwe Sodan
{achim.brucker, uwe.sodan}@sap.com

SAP AG
Central Code Analysis Team
Dietmar-Hopp-Allee 16
D-69190 Walldorf

Abstract: Static Code Analysis (SCA), if used for finding vulnerabilities also called Static Application Security Testing (SAST), is an important technique for detecting software vulnerabilities already at an early stage in the software development life-cycle. As such, SCA is adopted by an increasing number of software vendors.

The wide-spread introduction of SCA at a large software vendor, such as SAP, creates both technical as well as non-technical challenges. Technical challenges include high false positive and false negative rates. Examples of non-technical challenges are the insufficient security awareness among the developers and managers or the integration of SCA into a software development life-cycle that facilitates agile development. Moreover, software is not developed following a greenfield approach: SAP's security standards need to be passed to suppliers and partners in the same manner as SAP's customers begin to pass their security standards to SAP.

In this paper, we briefly present how the SAP's Central Code Analysis Team introduced SCA at SAP and discuss open problems in using SCA both inside SAP as well as across the complete software production line, i. e., including suppliers and partners.

1 Introduction

Software security is becoming an increasingly important differentiator for IT companies. Consequently, techniques for preventing software vulnerabilities during system development are becoming more and more important. Figure 1a shows the distribution of vulnerabilities since 1999 according to [Nat]. The majority of the vulnerabilities (e. g., most types of code execution, overflows, XSS, SQL injection), are caused by programming errors and, thus, counter-measures need to be taken on the implementation level.

Static Code Analysis (SCA), if used for finding vulnerabilities also called *Static Application Security Testing* (SAST), is a technique that analyses programs without actually executing them (i. e., in contrast to dynamic testing approaches) and, thus, can be applied in early stages of the development life-cycle where fixing of vulnerabilities is comparatively cheap [GK02]. Currently, SAP focuses on applying SCA for finding potential security vulnerabilities as early as possible. In principle, an SCA tool analyses the source code of a

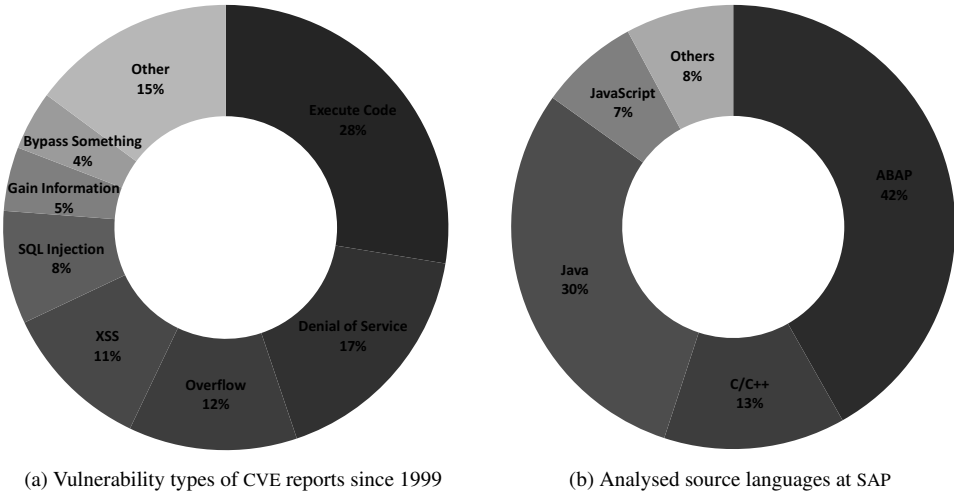


Figure 1: Overview of reported vulnerabilities and analysed source languages

software component (e. g., an application, library) and reports potential security problems: the set of findings. For each finding, an human expert is necessary to decide:

- If the finding represents a vulnerability, i. e., a weakness that can be exploited by an attacker (*true positive*), and, thus, needs to be fixed.
- If the finding cannot be exploited by an attacker (*false positive*) and, thus, does not need to be fixed.

Similarly, if an SCA tool does not report security issues, this can have two reasons:

- The source code is secure (*true negative*)
- The source code has security vulnerability but due to limitations of the tool, the tool does not report a problem (*false negative*).

In an ideal setting, the number of false positives and false negatives should be zero, i. e., the tool should be *sound* (i. e., no false positives) and *complete* (i. e., no false negatives). In practice, this cannot be achieved both due to fundamental reasons (e. g., undecidability of the halting problem) as well as compromises that need to be made between (potential) conflicting goals such as performance and correctness. Besides these, rather technical challenges, there are also non-technical challenges that make the introduction of SCA techniques difficult in a large development organisation with more than 12 000 developers.

In this paper, we report on our experiences in introducing SCA at Europe’s largest software vendor: SAP. In more detail, we motivate the choice for SCA based on SAP’s product security standard (Sec. 2) as well as present SAP’s code scan strategy (Sec. 3). Thereafter, we discuss our approach in introducing and integrating SCA into SAP’s software development life-cycle (Sec. 4) and discuss problems that we are facing (Sec. 5). Finally, we summarise our lessons learned and draw conclusions (Sec. 6).

```

<% DATA: ffs TYPE tihttpnvp, ff TYPE ihttpnvp.
DATA: lv_name type IHTTPNAM , " input parameter
      lv_value type IHTTPVAL . " input parameter
request → get_form_fields( changing fields = ffs ).
DELETE ffs WHERE name cs ' '.
LOOP AT ffs INTO ff.
  lv_name = cl_abap_dyn_prg ⇒ escape_xss_xml_html( ff-name ).
  lv_val = cl_abap_dyn_prg ⇒ escape_xss_xml_html( ff-val ). %>
  doc.writeln(' <input_type="hidden" _name="<%=ff-name_%"_value="<%=ff-val_%">">' );
<%
  CLEAR lv_name.
  CLEAR lv_val.
ENDLOOP. %>

```

Listing 1: An ABAP Business Server Page that contains XSS vulnerabilities. The input variables `ff-name` and `ff-val` are directly written into the DOM and, thus, an attacker that can influence these inputs can, e.g., inject arbitrary JavaScript into the web browser of the victim. Note the incorrect use of the sanitation function `escape_xss_xml_html` which does not sanitise the content of the variables `ff-name` and `ff-val`.

2 SAP's Product Standard "Security"

The overall goal of the software development life-cycle of SAP is to deliver secure, reliable, and high-performance software that is easy to use and fulfils the demands of SAP's customers. The non-product specific requirements such as usability, security, or performance are described in different *Product Standards*. The goal of SAP's *Product Standard "Security"* is to ensure that SAP products are secure by default as well as easy to operate securely. A prerequisite to achieve this goal is to ensure that SAP products are free of implementation related security vulnerabilities such as SQL injection, path traversal, memory corruption, or buffer overflow. Listing 1 shows an ABAP Business Server Page that contains a XSS vulnerability: user-influenced input from the input `ff-name` is written into the DOM without any checks. We refer the reader to the OWASP Top Ten [OWA13] or the CVE/SANS top 25 [Mit11], for a more comprehensive overview of vulnerability types.

Besides secure programming guidelines [SAP13b] and security training courses for developers and product owners, the use of Static Application Security Testing (SAST) (or, more general, Static Code Analysis (SCA)) is one of the key techniques that contributes to achieving this goal. Overall, SAST helps to detect implementation related security vulnerabilities in an early phase of the software development. In general, for finding implementation related security vulnerabilities, there are mainly two types of approaches (for a general introduction to SAST and competing approaches, we refer the reader elsewhere, e.g., [PCFY07, CW07]): 1. dynamic testing that analyses an application while the application is executed and 2. static code analysis that analyses the application without executing it. After an analysis of these techniques, SAP decided to introduce SAST as mandatory step for preventing implementation related security vulnerabilities. At SAP is mandatory (with the exception of technical reasons such as the use of technologies that are not supported by the available SAST tools) for products developed at SAP. The main reasons for this decisions are that SAST can be automated to a large degree, scales for large application (i.e., a large code basis), yields in repeatable results, does not require isolated test systems of

whole applications, and allows for fixing problems early in the development life-cycle. This decision is also backed up by our own studies in comparing, e. g., SAST and pen testing approaches as well as empirical studies done outside of SAP, e. g., [SWJ13]. Of course, SAST is only one building block of a holistic security testing approach [BB14].

3 SAP's Code Scan Strategy

The use of SAST at SAP is organised and governed by the *Central Code Analysis Team* that is responsible the code scan strategy. The code scan strategy is guided by the following four principles:

1. Security code scans are *embedded into the development process*, thus, should be as less disruptive for the developers as possible.
2. It is understood that there is always a compromise between accuracy and efficiency of the tools.
3. A manual review of the findings is necessary which requires human effort. The false positives are marked permanently so that manual review has to be done only once.
4. The tools will prioritise the findings and the development teams will concentrate on analysing and fixing all high priority findings; low priority findings will be spot checked, moved to the backlog and fixed as allowed by the capacities.

From these principles, the other responsibilities of the Central Code Analysis Team are derived. The two most important ones are:

- monitor the tool market, evaluate and select the tools that are used at SAP. If necessary, drive the development of own tools if no suitable tools can be identified.
- constantly monitor the efficient and effective use of the tools with the goal of developing and implementing improvements. These improvements include both the processes and the tools itself (including their configuration).

Thus, from an organisational perspective, this team acts as the first contact for security related code analysis (dynamic and static) for all stakeholders of SAP. From a technical perspective, the Central Code Analysis Team defines (and constantly improves) the default configurations of the tools, evaluates and introduces new tools or new versions of the tools.

If SAST cannot be applied to a product (e. g., it is written in programming language not supported by the a SAST tool), the development team has to develop a mitigation plan, i. e., describe the alternative techniques they apply to ensure the security of the product.

4 Introducing Static Application Security Testing at SAP

In this section, we briefly introduce the software development life-cycle at SAP and explain the processes that ensure an efficient and effective use of SAST by the development teams.

4.1 Integrating SAST into the Software Development life-cycle of SAP

The software development at SAP follows an agile and decentralised approach, i. e., the development teams are empowered to organise the details of their development process as long as they achieve requirements of the various products standards of SAP. The compliance to these standards is reviewed at globally defined milestones, called *Quality Gates*. Moreover, the quality gates allow to synchronise the various sub-process of the software development life-cycle at SAP, called *Idea-to-Market* (I2M) (see Figure 2):

1. The *Idea-to-Portfolio* (I2P) process describes how ideas from different sources inside and outside SAP are collected, matured and turned into a solution portfolio.
2. The *Portfolio-to-Solution* (P2S) process describes the path from given portfolio decision to the customer available product. This includes the definition of the product scope, release of the budget, as well as the actual development of the product. This process contains three sub-processes: *Planning and Setup*, *Development*, and *Release Preparation* and the quality gates are:
 - (a) *Portfolio-to-Planning* (P2P): Among others, this quality gate includes an approval of the actual business case.
 - (b) *Planning-to-Development* (P2D): Handover from the planning teams to the development teams which, among others include the commitment that the necessary development resources are available.
 - (c) *Development-to-Production* (D2P): This quality gate defines the end of the development of a specific product version. After passing this quality gate, the software is either prepared for on premise delivery or prepared for hosting. Here, the product groups need to prove at this point in time that they comply with the Code Scan Strategy, i. e., scanned and analysed their product and, more importantly, fixed the identified (and prioritised) vulnerabilities.
 - (d) *Production-to-Rollout* (P2R): After passing this quality gate, the solution is finished to be rolled-out to customers. Before the solution is rolled-out, the *Quality, Governance, and Production Team* validates, again, that the solution complies with the various products standards at SAP. For security, e. g., this includes a penetration test of the finished product.
3. The *Solution-to-Market* (S2M) process describes the necessary steps to bring a developed solution to the market.

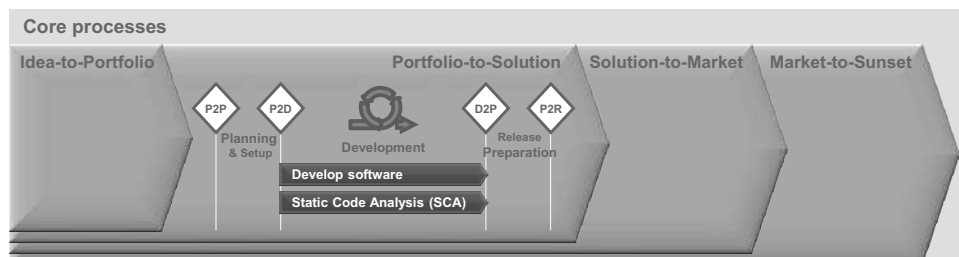


Figure 2: The core of the software development life-cycle at SAP

4. The *Market-to-Sunset* (M2S) process defines the controlled end-of-life of a solution as well as the maintenance of the solution.

As SAST helps to prevent vulnerabilities caused by bad programming, i. e., on the implementation level, we concentrate our SAST effort on the Portfolio-to-Solution (P2S) phase. As we will see in the next section, customer feedback received in the Market-to-Sunset (M2S) phase plays an important role for improving and focusing the use of SAST at SAP.

4.2 Continuous Improvement of our SAST Approach

Introducing SAST is not a one-time effort: only the constant monitoring and improvement of both the tools as well as the processes can ensure the *efficiency* and *effectiveness* of SAST as technique for preventing vulnerabilities. The main goals are increasing

- the technological scope, i. e., developing support for new programming languages or development frameworks.
- the security scope, i. e., by developing checks for new vulnerability types.
- the efficiency by lowering the false positive and false negative rates. Similarly, the prioritisation of the findings needs to be monitored and improved.
- the usability of the tools, e. g., develop better integration into the develop environments or improving the response time of central servers supporting scans or audits.

Measuring the efficiency and effectiveness of SAST is difficult, defining and measuring good key performance indicators (KPIs) is even more difficult. Besides recording the basic data (number of issues found, time required for fixing issues, etc.), we concentrate our efforts for improving our SAST approach on direct collaborations with various stakeholders:

- the *Product Security Response Team* which is the main contact point for customers and external security researchers reporting security issues of SAP products and, thus, manages the process for analysing the root cause of all externally reported vulnerabilities and ensures that they are fixed in a timely manner.
- the *Quality, Governance, and Production Team* which validates (e. g., using pen tests) before shipment that SAP's products comply to the product standard "Security."
- the *Development Teams*, including their security experts, that execute the processes defined by the Code Analysis Team, i. e., they actually use the various SAST tools.

Besides the constant improvements of the usability, together with the development teams, we are concentrating on improving the efficiency of the tools by

- *Reducing the number of false positives*: False positives are detected by the development teams (together with their security experts) during the audit and might, e. g., caused by new security APIs not yet recognised by the scan tool. Whenever they recognise changes in the number of false positives (or have other suggestions for improvements) they should contact the central code scan team and collaborate in improving the current tool configuration (or search for better alternatives).
- *Reducing the number of false negatives*: False negatives are much harder to detect, as the tool does not report them. To improve in this area, we concentrate on vulner-

Table 1: SAST Tools used at SAP

Programming Language	Tool	Vendor
ABAP	CVA (SLIN_SEC) [SAP13a]	SAP
C/C++	Coverity [Cov13]	Coverity
All other languages	Fortify [HP13]	HP

abilities reported by external researchers as well as vulnerabilities found internally by alternative security testing approaches (e. g., the validation done by our Quality, Governance, and Production Team).

For each found vulnerability, we check if it could, potentially, be detected by a SAST tool. If it could be found, we analyse (together with the stakeholders) why it was not detected during the regular SAST efforts and how we can improve the situation.

The outcome of this analysis can be that the vulnerability

- is not detected by the most current configuration. In this case, we check (together with the tool vendor) how the configuration or tool can be improved to report this issues in the future.
- was reported by the default cans but not fixed prior to shipment. In this case, we investigates the reason for not fixing the issue. This might result in an updated version of the tool configurations (e. g., re-prioritising certain issues types).
- The vulnerability was reported for a product release that was developed before the code scan strategy was implemented. In this case, we check if the vulnerability is detected by the current implementation of the code scan strategy.

Moreover, for products that were not scanned during development, the number of reported vulnerabilities that can be detected by SAST is an additional argument for justifying the efforts of implementing the Code Scan Strategy.

The effort in improving the tooling is prioritised according the size of the code base. ABAP is still the most used language at SAP (see Figure 1b). These priorities can change rather quickly: a few years ago, JavaScript was only used rarely. During the last two years, the use of JavaScript, both client-side and server-side, increased significantly and JavaScript is, in the near future, about to replace C/C++ as the third most used language at SAP.

4.3 Where We Are Today

The implementation of SAP's code scan strategy requires over 2 000 developers and security experts to work, on a regular basis (scan and audit), with SAST tools—even more developers are involved in fixing vulnerabilities found.

SAP uses three different SAST tools (see Table 1) across all development units and the whole product range (ranging from mobile apps to on-premise offering to on-demand offerings). Until today, more than two billion lines of code were scanned and the findings analysed and, if exploitable, fixed. To customers, the most visible result of implementing a rigorous

code scan strategy was the Security Patch Day in December 2010 where SAP published over 500 patches (called *SAP Security Note*) [SAP10]. This, of course, requires a careful planning and agreement with customers. In the meantime, patches fixing vulnerabilities detected by SAP internal code scans that are released using regular support packs. This reduces the effort that customer require to apply these patches significantly.

For Coverity and Fortify, the central code scan team operates a company-wide server infrastructure for executing and archiving code audits. Close to 3 000 software projects (each of them having multiple versions) are stored on these central servers. For ABAP the scans are deeply integrated into the development tools (e. g., the ABAP Test Cockpit) and, thus, no additional central infrastructure is necessary.

To ensure the security across the whole software supply chain, SAP makes also use of external services (e. g., offered by Veracode [Ver13]) for analysing third-party code that is not available as source code. Moreover, we regularly evaluates alternatives to the tools used (e. g., Checkmarx [Che13] or IBM Security AppScan Source [IBM13]), e. g., to increase the number of supported programming languages.

5 Open Issues

Low security awareness and hard to estimate risk of not fixing security issues. Even though SAP has rolled out mandatory security training courses for all developers since 2011, the security awareness during regular development is insufficient. Moreover, it is often hard, if not impossible, to estimate the potential financial risk of not fixing a security issues. Thus, particular, in cases where development efforts need to be prioritised between the investigation security issues and implementing new (functional) features, teams try to minimise the effort spend for analysing or fixing security issues.

Pushing SAST across the software supply chain. To increase the software security of SAP's offerings, we cannot concentrate our efforts on SAP's software development: we need to include suppliers and partners to ensure the security of

- third party software (including commercial offerings, Freeware, and Free and Open-Source Software) used in the development,
- partner products sold by SAP or used by SAP customers together with SAP solutions.

Finally, our customers start to impose similar requirements to us. Thus, we needs to be able to provide a proof of our security initiatives, e. g., the successful execution of our code scan strategy.

Huge and hybrid multi-language applications. Many modern applications are built by composing many different components that are not necessarily written in the same language. For example, a Software-as-a-Service (SaaS) offering might be based on a user-interface written in JavaScript and a backend that uses both a Java and ABAP stack. Thus, large applications statically does not only raise the question of how to modularise (i. e., al-

lowing for scanning each component in isolation while still allowing for a precise analysis of dataflows across several components) scan. It does as well raise the question of how to analyse dataflows across different programming and framework contexts.

Dynamic programming paradigms and languages. Modern applications are highly distributed and often based on asynchronous and dynamic programming concepts and languages, e. g., JavaScript. These large (millions lines of code) JavaScript applications can neither be handled satisfactory with state-of-the-art static nor with dynamic approaches.

Lack of standardised regression test suites. New versions of SAST tools (as well as alternatives tools) can have a significantly different behaviour (e. g., not reporting issues that were reported previously, reporting a large number of false positives). Thus, we requires an effective and efficient way for evaluating (testing) security testing tools. This includes artificial test suites such as the SAMATE test suites [SAM13] (only available for Java and C/C++) as well as real products that use different programming styles. Building and maintaining such test suites is non-trivial and labour-intensive. Similarly, test methods for the rigorous testing of technique for SAST configurations (e. g., cleansing rules) as well as security functions (e. g., cleansing or sanitation functions) needs to be developed.

6 Conclusion and Lessons Learned

While Bessey et al. [BBC⁺10] present the perspective of a vendor (i. e., Coverity) of a static code analysis tool “a few billions lines of code later,” we discussed the users perspective based on analysing several billions lines of code.

In our opinion, applying SAST is a never-ending journey requiring constant monitoring and improvements. Thus, while there are still many open questions (recall Sec. 5), we believe that, based on our experiences, we can already provide valuable insights and recommendations on how to introduce and improve SAST in an agile software development life-cycle. To begin with, we recommend to follow the five keys to success identified by Chandra et al, that we briefly (see [CCS06] for details) recite here:

1. *Start small.* Work closely with one group to sort out any kinks in the adoption process, and once this pilot group succeeds, move on to replicate that success.
2. *Go for the throat.* Rather than trying to stomp out every security problem on the first day, pick a handful of things that go wrong most often and go after them first.
3. *Appoint a champion.* Find a developer who knows a little bit about every part of the system and likes to mentor. Put your new champion in charge of the tool adoption.
4. *Measure the outcome.* Tools spit out quantifiable results, so use them to track projects over time. Code analysis results can indicate where trouble is brewing, but it can also tell you whether your education and training efforts are paying off.
5. *Make it your own.* Use the tool to enforce your own standards and guidelines. Spend some time investigating customisation capabilities to achieve deeper inspection and maximised accuracy for your environment.

In our experience, these five keys are only the first step and we suggest to at another seven and, thus, to complete the dozen:

6. *Plan and invest enough resources.* Introducing SAST requires significant resources, not only for analysing and fixing findings. You need also resources for educating developers, operating tool infrastructure. Finally, you might need to introduce new expert roles (e. g., security experts) to your organisation.
7. *Plan and invest enough infrastructure.* Providing a reliable, fast, and scalable solution for executing the scans as well as storing and managing the results requires a significant amount of hardware. Depending on the tool (e. g., if local audit and scans are supported), not only powerful servers might be required, also an upgrade of the development machines of the developers might necessary.
8. *Do understand your developers as you allies.* While some processes suggested by SAST vendors suggest that developers and security experts are fighting against each other, we strongly recommend to understand team as one team fighting for a secure product: SAST should be a tool that supports your developers in producing high-quality software and, thus, should be introduced in collaboration with them.
9. *Execute scans regularly.* SAST is not a one-time effort that solves all security issues on-and-for-all. The efficient and effective use of SAST requires its deep integration into the development process and tools. Code scans should be executed and monitored regularly—transferring audit results from one scan to the next works best for small changes of code basis. Thus, apply code scans nightly or at least weekly.
10. *Plan your changes and updates.* Do not introduce changes in the scan infrastructure (new versions of tools or updated to their configuration) within one development cycle. The change in results that might be caused by this may reduce the confidence of the developers into the tool. Always plan for such changes ahead and introduce them together with your developers.
11. *Do get support (and commitment) from your management.* Introducing SAST is a significant investment that requires long-term support from management (recall the memo from Bill Gates [Gat02]). While investment into security often needs to be balanced with other investments, it must be clearly understood by all stakeholders (including management and developers) that SAST can only improve the product security if and only if it is understood as a continuous long-term investment.
12. *Do not stop here.* Introducing SAST can only be the first step. SAST needs to be complemented with other techniques (e. g., threat modelling, pen testing) and, thus, needs to be embedded into a *holistic security testing strategy* [BB14] which, in itself, should be part of a secure software development life-cycle.

This dozen keys to success need to be implemented by clearly defined processes and supported by the responsible roles. Watch out, these keys are necessary but not sufficient for the successful introduction and execution of SAST in a large software development unit.

Overall, static code scans are useful and contribute to avoid vulnerabilities that, otherwise, would be found by external researchers and, thus, could harm the reputation of SAP.

References

- [BB14] Ruediger Bachmann and Achim D. Brucker. Developing Secure Software: An Holistic Approach to Security Testing. *Datenschutz und Datensicherheit*, March 2014.
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.
- [CCS06] P. Chandra, B. Chess, and J. Steven. Putting the tools to work: how to succeed with source code analysis. *Security Privacy, IEEE*, 4(3):80–83, may-june 2006.
- [Che13] Checkmarx. Checkmarx. <http://www.checkmarx.com/>, 2013. Site visited on 2013-12-07.
- [Cov13] Coverity. Coverity. <http://www.coverity.com/>, 2013. Site visited on 2013-12-07.
- [CW07] Brian Chess and Jacob West. *Secure programming with static analysis*. Addison-Wesley Professional, first edition, 2007.
- [Gat02] Bill Gates. Memo from Bill Gates. <http://www.microsoft.com/en-us/news/features/2012/jan12/gatesmemo.aspx>, January 2002.
- [GK02] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report Planning Report 02-03, National Institute of Standards & Technology, May 2002.
- [HP13] HP. HP/Fortify. <http://www.hp.com/us/en/software-solutions/software-security/index.html>, 2013. Site visited on 2013-12-07.
- [IBM13] IBM. IBM Security AppScan Source. <http://www-03.ibm.com/software/products/en/appscan-source/>, 2013. Site visited on 2013-12-07.
- [Mit11] Mitre. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. <http://cwe.mitre.org/top25/>, 2011. Site visited on 2013-12-02.
- [Nat] National Institute of Standards and Technology (NIST). National Vulnerability Database. <http://nvd.nist.gov/>. Site visited on 2013-11-24.
- [OWA13] OWASP. OWASP Top Ten. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013. Site visited on 2013-12-02.
- [PCFY07] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.
- [SAM13] SAMATE. SAMATE Reference Data Set. <http://samate.nist.gov/SRD/testsuite.php>, 2013. Site visited on 2013-12-07.
- [SAP10] SAP. SAP Note 1533030: Patch Day 12/2010: General Info for SAP Suite and NetWeaver, 2010.
- [SAP13a] SAP AG. SAP NetWeaver Application Server, add-on for code vulnerability analysis. <http://scn.sap.com/docs/DOC-48613>, 2013. Site visited on 2013-11-24.
- [SAP13b] SAP AG. SAP Secure Programming Guides. <http://scn.sap.com/docs/DOC-48612>, 2013. Site visited on 2013-11-24.
- [SWJ13] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: a controlled experiment. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, pages 1–10. IEEE, November 2013.
- [Ver13] Veracode. Veracode Independent Software Audits. <http://www.veracode.com/services/application-audits.html>, 2013. Site visited on 2013-12-07.