SAVE - Static Analysis on Versioning Entities

Bastian Braun
University of Hamburg
Department of Informatics
Hamburg, Germany
braun@informatik.uni-hamburg.de

ABSTRACT

Insufficiently tested software releases provoque a competition between 'exploiters' versus 'patchers'. Developing secure software from scratch greatly reduces maintenance effort. The integration of regular security checks combined with patch proposals at development time enhances the system's usability and software quality. This paper presents a software development system including version control, security analysis and patching support. As a practical aspect, avoiding flaws becomes easier even for non security experts.

Categories and Subject Descriptors

D.2.7 [**SOFTWARE ENGINEERING**]: Distribution, Maintenance, and Enhancement—*Version control*
; D.2.9 [**SOFTWARE ENGINEERING**]: Management—*Software quality assurance (SQA)*; K.6.3 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Software Management—*Software development*
; D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques; D.2.1 [**SOFTWARE ENGINEERING**]: Requirements / Specifications—*Methodologies, Tools*; C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems—*Client / Server*

General Terms

Security

Keywords

Version Control System, Static Analysis, Incremental Analysis, Secure Software Development

1. INTRODUCTION

Before releasing a new software product, it is advisable to test the code for any kind of faults. Several methods exist to find security relevant bugs. However, analysing thousands of lines of code is not only time-consuming, but the number of expected warnings makes it hard to distinguish between real faults and false positives.

Software developers are familiar with this problem. An in-depth analysis of all warnings raised by automatic analysis tools is almost as costly as the development. Time pressure makes manufacturers release their software without sufficient security checks. In particular, the application of analysis methods as the final step — if at all — leaves no time for exhaustive fault correction.

In the majority of cases, the management sets the date for final release based on marketing or productivity aspects. However, the software developers will have to meet the deadline. Additionally, according to Murphy's law ("If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way."), undesirable events happen in the situations where you need them the least. As a result, security analysis and patching are delayed after official product deployment. This is one reason why most updates are issued during the first year. This rule applies for commercial as well as for free software.

To transparently integrate security analysis methods in the development process would incur a tremendous improvement of this situation. Most errors can be prevented in an early phase before they impact bigger parts of the code. However, due to the integration of security analysis, the software development process has to be slightly modified. Beside the analysis, a patching step has to fit in the workflow. By experience, we can state that every change in the habitual process is blamed at first. So, the advances have to fit well in the used proceeding. The results of any security analysis have to be prepared for easy understanding. Graphical or examplified hints are most helpful. We strive to avoid an overwhelming number of warnings as these would lead to motivation loss on the developer's side. In particular, the number of false positives is to be minimized. Finally, a good approach takes the actual infrastructure into account and makes use of standard development software.

Contributions

How can the regular analysis be done? We propose a flexible and portable approach which integrates into server-side version control systems. There are implementations that fit in the Integrated Development Environment (IDE) of a developer, e.g. Coverity, Fortify, Klocwork. An analysis can be run before submitting the code to the version control system. In this paper, we present another approach:

- We present a centralized server-based approach to reg-

SESS'08, May 17–18, 2008, Leipzig, Germany.

ularly run analysis algorithms, e.g. static analysis. (Section 3)

- We propose a system that is able to manage code versions, addresses the responsible developers for a vulnerability and finally proposes ways to patch it. (Sections 4 and 5)
- We show the advantages of this approach in comparison to the distributed client-based solution. (Section 6)
- We identify innovative research directions for future work. (Section 8)

2. BACKGROUND

There are numerous types of software development models. Each of them describes a sequence of phases. One phase consists of some kind of actions, e.g. define the requirements, make an analysis, design the program, implement the program, test, and operate.

The models commonly schedule security tests at the end, if at all (cf. waterfall model [4], spiral model [1]). In practice, the first phases, and in particular the implementation phase, take more time than provided by the timetable. For example, one has to make an adjustment of the software according to changing need. In other cases, unforeseen problems occure, e.g. flaws in interface design, sick employees, hardware crashes, urgent interrupts caused by other software, or just individual mistakes. Assume, for example, that a big bug of a previously released product is reported. The resources of the current project are needed to fix it. This however delays the development which in turn leads to increased time pressure.

As an impact, the last phase must be abbreviated. By this, however, more faults occure and additional effort for fixing and patching will be necessary in the future.

The aim of our approach is to enhance existing software development strategies instead of inventing a new strategy. We propose the integration of security checks into the software implementation phase. However, these cannot replace exhaustive checks in the last phase as the applicable methods for checks during development are limited. Dynamic testing methods like fuzzing require executable programs for testing. This assumption cannot be made during development. Static analysis is an approved method for security testing. It analyses even incomplete source code and delivers sufficiently good results. We suppose static analysis as the security checking method.

Basically, we assume the following scenario. The whole group of developers is divided into teams. A dedicated security team would be beneficial but our approach does not rely on it. The project consists of several files and modules. Each team works on a module of the software. We suppose that numerous developers edit the same file at the same time. A version control system (VCS) manages synchronisation issues.

3. SERVER-SIDE SECURITY ANALYSIS

We tackle the problem described in the last section by partially integrating the work of the abbreviated checking phase (cf. Figure 1) into the implementation phase. Checking can be done in background by static analysis tools. In contrast

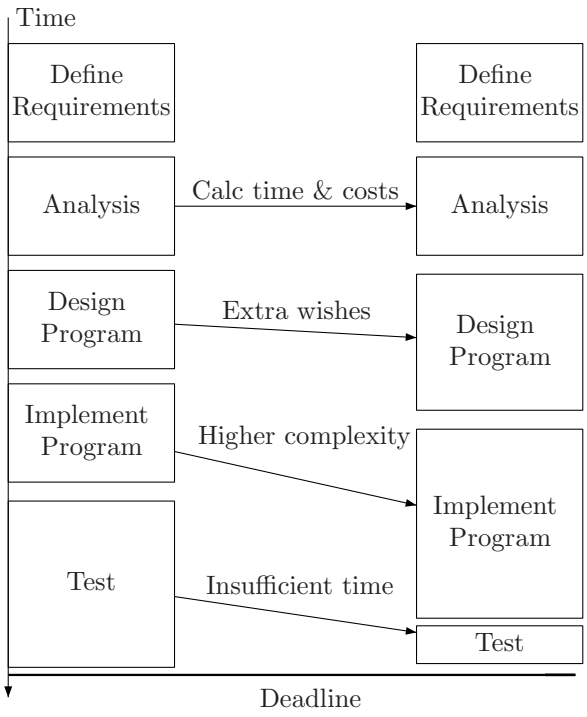

Figure 1: Examplified sequence of phases in software development. The left column represents the process as intended at the beginning, the right column displays the real life situation.

to the approach of checking locally on the users' side, we propose centralized checking on the server side after each commit to the repository.

Let us assume three developers working on a module and sharing a VCS repository (cf. Figure 2).

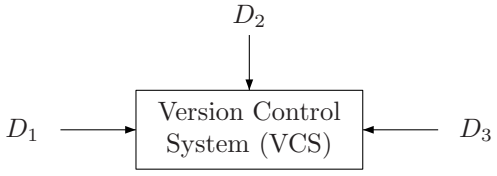

Figure 2: Exemplary development scenario with three programmers working on one module and sharing a versioning repository.

Each of them modifies the code by writing to the local working copy and committing the changes to the VCS. The VCS then uses the differences of the previous version and the current version to run static analysis checks with respect to the changes, i.e. the system seeks all new data- and controlflows to find security relevant faults in these. There are algorithms for this kind of analysis, called incremental analysis (cf. Section 7, [5, 11, 12]). So, given the nodes of the control flow graph that are affected by the recent changes, we consider only paths passing such nodes. Assuming that node n_i has changed, then all paths $n_1, \ldots, n_i, \ldots, n_j$ through the control flow graph need to be checked while the other paths (not containing n_i) can be left out. Additionally, it is possible to limit the set of monitored variables. Assuming that

all paths in the previous dataflow graph are acceptable, it is sufficient to look into modified dataflow paths. An appropriate method is called *program slicing* [7] meaning that we reduce the code to the significant lines. We do not consider the lines not contributing to the vulnerable computation. That way, we do not take all variables into consideration.

Let

```
a = a - 1
```

be a line in the code and D_1 modifies the code to

```
if (a > 0)
a = a - 1
else
a = a + 1
```

Then, the control flow graph has changed and the new control flow branches are inspected. In the initial case, any negative value is eventually taken by `a`. The assumption must be abandoned which can lead to unforeseen problems depending on the rest of the code, e.g. an exit condition could become true when `a` becomes -5. So, the program would not terminate in this case. If the analysis tool raises a warning of an infinite program run, then D_1 receives a notification that she needs to review these lines and eliminate the fault. Therefore, every line of code is tagged with the responsible developer, i.e. the one who inserted the line or the one who edited the line the last.

We propose to run the security analysis centralized unlike other approaches that suggest distributed checking on the developers' clients (cf. Section 6). One can figure the server side checking tool as an additional user V who always takes the last version and the changes, locally checks the code, and finally submits the results to the responsible developer D_i (cf. Figure 3). D_i in turn fixes the bug and submits her changes to the repository where V checks out again and scans for faults. This process is iterated unless no fault is found in the current version.

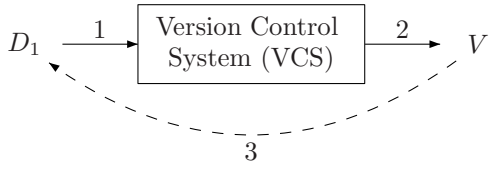

Figure 3: Examplary development scenario with an additional user as a verifier. After the developer submitted new code (1), V checks current version out (2), runs incremental analysis and sends feedback to D_1 (3).

4. CODE ATTRIBUTION

If a potential fault is localized, it is important to address the responsible developer. In most cases, it is not possible to determine a unique appropriate one. Then, all candidates have to be addressed. However, we must not contact too many as they will feel annoyed in a while. What we need is a link from the vulnerable code to the responsible developer. Therefore, the VCS assigns every line of code to one developer. We obtain a code attribution function, mapping the lines of code to the set of developers.

Definition 1. Let C be the set of all lines of code generated by the software project in focus. If D is the set of all developers, then CA is the corresponding *code attribution function* $CA : C \mapsto D$, $CA(c_i) = d_j$, iff the following holds: for every line of code c_i, the developer d_j inserted this line and it has not been changed or d_j changed this line the last.

For a subset $C' \subseteq C$, $C' = \{c_i, \ldots, c_{i+k}\}$, holds

$$CA(C') = \bigcup_{l=i}^{i+k} CA(c_l)$$

.

Generally, a bug does not happen in one single line but in a series of lines leading to the failure. Given a set X of lines, the set $CA(X)$ of developers is informed about the flaw. As we will see in Section 5, it is important, that the related patching submit can be identified.

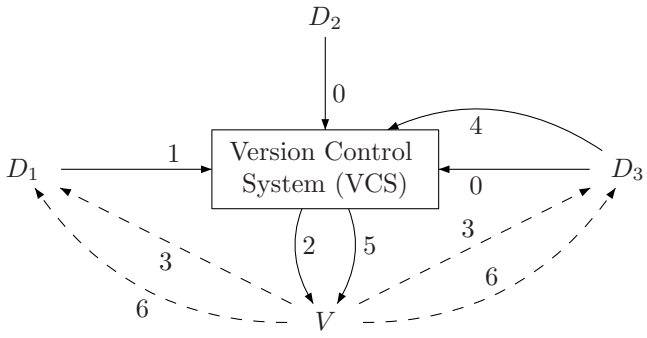

Figure 4: Examplary development scenario with additional developers. After developers D_2 and D_3 first submitted their code (0), D_1 submits next (1). Verifier V checks out (2) and detects that a flaw occurs on the code of D_1 and D_3. She notifies them (3), and D_3 submits a patch (4). V checks out the patched version (5), finds no fault and informs D_1 and D_3 (6).

4.1 Example

As an example, assume that three developers are involved in the project (cf. Figure 4). D_2 and D_3 submit their contributions respectively. At this point, no fault is detected yet. Next, D_1 submits her code, and the analysis delivers a warning. The path that leads to the possible flaw contains both code of D_1 and code of D_3. So, both developers receive a notification with the fault description. Though the fault is detected after D_1 submitted, D_3 is responsible. For instance, she does an implicit variable type cast leading to an undefined value on the code of D_1. After fixing the bug, D_3 has to submit the patched code that is checked by V. Finally, V informs D_1 and D_3 about the successfull patch.

4.2 Patch Management

We now want to explain how the patched code is inserted into the project.

Therefore, we first assume that the vulnerable code is still the current code version (i.e. no changes have been applied yet). So, the developer can submit her changes, and the new code base is checked. A more realistic scenario is the following (cf. Figure 5). Some new code is submitted, the security check starts, more code is added, the checking finally ends

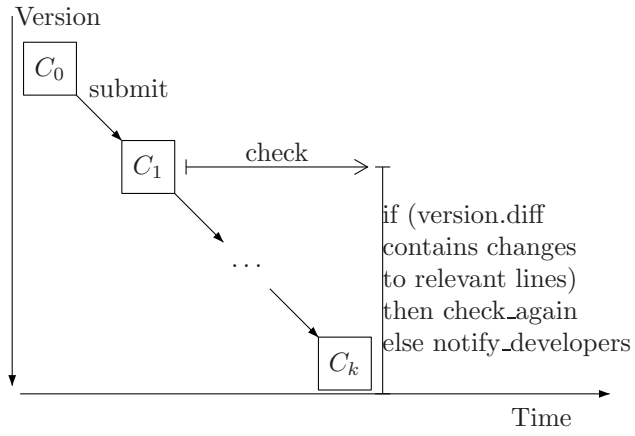

Figure 5: Visualisation of ongoing development and security checks. Some flaws are introduced in version C_1. They are detected, i.e. the checking run finishes, when version C_k is the last submitted version. The responsible developers are notified iff their code has not changed.

and outputs a list of warnings etc. These results rely on an outdated code version. The version control system then has to look into the recent changes to find out if the problems may still exist. If this is the case, the responsible developers are notified to review the code and fix the problems. Afterwards, the patches are merged with the current code version to be checked (cf. Figure 6).

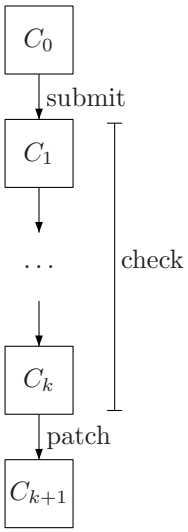

Figure 6: Visualisation of patch inclusion. Some flaws are introduced in version C_1. They are detected, i.e. the run finishes, when version C_k is up-to-date. The responsible developer submits a patch that is treated as the next version.

Of course, a notification to the authors of the code is not reasonable if the code in focus has changed and the detected fault cannot occur in the current version any more. However, as the checks just rely on recent changes and do not test the whole code, they finish sooner. That is why we can assume that the number of new submits and the amount of intermediate changes is low. The actually submitted changes depend on the number of developers, and if they have a submit ratio policy.

In the case of a fault, the analysis tool obtains a vulnerable line of code or a set of lines that cause the failure. By checking if the fault can still occur, we mean testing if at least one of these lines is still vulnerable. Therefore, we perform threefold tests:

1. Regarding the differences of code, if none of the lines leading to the bug has changed compared to the checked version, the warnings are delivered to the responsible developers.

2. Consider the control flow that is taken in the fault scenario. If the control flow path which leads to a vulnerable line of code does not contain this fault any more, the warnings are discarded.

3. Check if the bad dataflow can still occur, i.e. if the malicious path through the dataflow graph exists. If it does, deliver the warnings, else discard.

We use incremental analysis to efficiently check the changes in the control and dataflow graphs (cf. Section 7). First, we check the differences to detect the trivial case that no changes have been applied to this part of the code. Next, we exclude relevant changes to the control flow graph that rule out this fault. Finally, we test if the bad dataflow has been diverted.

4.3 The Conservative Approach

There is a more conservative approach in case of higher security demands.

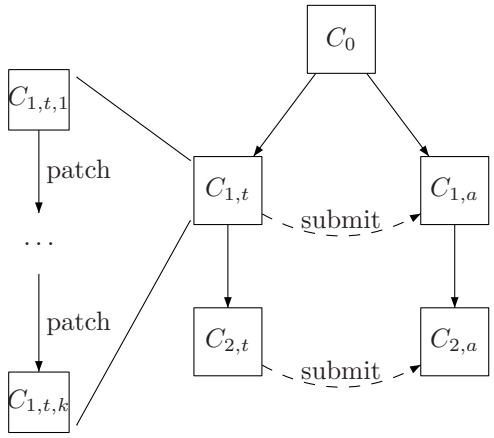

Figure 7: The conservative approach withholds new submits until the previous one is clean. When all flaws are removed the clean code is treated as a new submit to the approved branch.

We create two development branches, one for testing versions and one for approved versions (cf. Figure 7). Given an initial version, the system proceeds as follows. The initial version is tested. If there is a commit during the test, it is merged to the testing branch. The same holds for further submits. When the test of the initial version finishes, we distinguish between two cases. If the code is clean, i.e. no

faults are found, the next submit from the testing branch is scanned. Otherwise, the notification and patching process is started. All warnings are delivered to the responsible developers until the code is flawless. Then, the code is merged to the approved branch. The next submit is checked and so on. That way, we derive a branch which contains only checked and clean versions.

5. LEARNING BY DOING

As already mentioned in Section 4, it is important that patches are distinguishable from new code submits.

First, the developers who are being informed but finally turn out to be not responsible need to know that the patch has been done. So they do not need to review the code.

Secondly, we aim at building a "learning" system. Roughly speaking, we create a patch plugin P that gets to "know" the bug as it receives the analysis report. It informs the developers and gives additional information, for example the affected lines of code and which kind of fault is meant. Due to the patches, developers apply over time to the software project under version control, a history of vulnerabilities and appropriate patches can be archived. Based on this history, P can recognise flaws and propose an appropriate patch to the developer. However, it is very unlikely to find exactly the same fault later on. We need a vulnerability pattern matching. In this way, we can classify the recorded faults and determine the class of newly detected faults. In doing so, we derive the patches applied to recent faults in the same class. These patches serve as patch propositions in the future. The developer adjusts them and thus applies approved patches.

5.1 Fault Classification

For the classification of faults, we actually take two approaches. We suppose the first to be more complex and more precise. Assume that the error occurs in a line containing the statement S at procedure $Proc$. Without loss of generality, we assume S to be of the form

$$y := f(x_1, \ldots, x_n)$$

where $n \geq 0$ and f denotes an arbitrary function. The set of interesting variables is denoted by I. We want to distinguish between the relevant lines for the fault and the other lines. The latter will not be considered for the classification. Important lines have an impact on the computed term, i.e. they are needed for the computation of the x_i. So, initially $I = \{x_1, \ldots, x_n\}$. Next, we start a backwards analysis. For each line

$$z := g(u_1, \ldots, u_m)$$

where $m \geq 0$ and g denotes an arbitrary function, we distinguish between two cases:

1. If $z \in I$, then $I = (I - \{z\}) \cup \{u_1, \ldots, u_m\}$ and record the line,
2. else ignore the line.

On arriving at the first line, we reduced the code to the lines leading to the vulnerability. This process is sometimes called "program slicing" [7]. Finally, we build the program representation graph [10] of the remaining vulnerable procedure, i.e. without ignored lines. We apply *the Sequence-Congruence Algorithm* [10] on an intermediate simplified representation of the code because the algorithm supposes a limited programming language. For example, the GNU compiler collection tool (gcc) produces GIMPLE code [2] as an intermediate step at compiling. GIMPLE fulfills the requirements of the algorithm.

For every vulnerability, we build the program representation graph. If there is a class of isomorphic graphs, the corresponding patches are proposed, in the other case (if there is no such class) we create a new class.

The other more naive approach means building the classes only on the basis of the fault statement. This statement does not need to be the cause of the flaw but the fault happens in this line. So, we suppose that faults occuring in a similar statement can be sanitised in a common way.

We will evaluate both approaches and decide for the best. By these examples, one can see the conflict of precision in classification. In general, we derive more appropriate patches if we accept more complex classification. The intent is to find the best level of precision.

5.2 Patch Evaluation

One step forward is a quality control of the patches. If, for instance, a bug is reported, and the developer submits a patch which either does not remove this bug or introduces a new bug, this malicious patch will be proposed in the future again and again if it is not marked malicious. We give two solutions to the problem. The first and surely most accurate solution is engaging a security team for comparing primal and patched code with respect to the detected bug. They finally decide on the quality. This team could even submit their own patch where necessary. The second and more technical solution is monitoring the patched lines of code. If after submitting the patch, another fault occurs on the same lines, the patch is considered being amiss. The determined quality of the patch will be part of the proposal to the developer.

5.3 False Positives

Not every warning is a real flaw. If a warning turns out to be a false positive, the developer can mark it as such. These labellings are stored on the server. The same warning can thus be suppressed in the next report. That way, we avoid annoying the developers with known false positives. If however any line of the respective sliced program changes, the labelling must be removed as the false positive may have turned into a true positive.

We want to make sure that no true positive is ignored and so accept the occurrence of false positives in case of doubt. Then, however, we can label them as false positive candidates.

6. COMPARISON WITH THE DISTRIBUTED APPROACH

The centralised approach proposed in this paper implies the execution of security checks on the server side. In contrast, the distributed approach stands for running analysis on the hosts. The last however suffers from some drawbacks in comparison to the first as we will show.

- The analysis on the server can be divided into smaller steps. The server takes every submit and can thus use these smaller portions of changes to run checks faster.

The host has to receive all changes made since the last checkout and thus runs for a longer time.

- The host is kind of blocked during the run. Though the developer could go on working she will not be able to submit her code till all warnings are smoothed out. This way, even the contribution of error-free code is delayed.
- The same contribution is checked repeatedly, i.e. every developer has to check all code for every submit. Assume a developer submitting her code. She first has to check out the current version and merge her contribution. Then, she runs the analysis and after removing all possible flaws, she tries to submit again. If another developer successfully submitted new code during her analysis run, she needs to rerun. This scenario can happen on every host at every time.
- Responsibility assignment is hard in distributed scenario. For example, consider the case that a fault occurs during a local check but not in newly added code. Obviously, the new code caused the detection but the error occurs on other code. It is not trivial to determine who should be addressed for fixing.
- We observe a diversity of Integrated Development Environments (IDEs) on the developers' side, whereas there are much less applications on the server side. What matters is not only the use of different IDEs in the same organisation but the variety throughout the world. As an automatic integrated security checker has to interact either with the developer's IDE or with the VCS on the server, standardisation and manageability are in favor.
- The management of false positive reports can be done best on a central entity. Given a distributed labelling, consistency problems are very likely. Despite this, developers cannot benefit from the others' labelling without synchronization. The centralised approach schedules the server for synchronous label and report management.
- It is easier to enforce a security checking policy on a single server than on each developer's system. Let us consider a developer who is irritated by lots of warnings. No doubt that she tends to discard each of them. This behaviour is easily traceable on a centralised system.

Nonetheless, the distributed approach has its advantages, too. For instance, a security check right after submitting on the developer's host gives fast feedback. Of course, the server's computation and performance requirements increase in the centralised case. However, this should not pose a problem regarding general hardware needs. The server's role is not changed by its new challenge as it is a single point of failure in either way. So, additional precautions must be taken.

7. RELATED WORK

Schreckling and Johns propose an automatic code-audit tool named CISAT ("Combination and Integration of Static Analysis Tools") [6]. It is meant for easy integration into IDEs and VCS's. This tool manages the execution and presents the results of numerous static analysis tools.

The selection of favorite tools can be made on the basis of qualitative comparison. For instance, Wilander and Kamkar give a detailed analysis of some static analysis tools [9]. All these tools claim to detect buffer overflow and format string vulnerabilities. Though the false positive rate was high, they summarise that "the main usage for these tools would be as support during development"[9].

Neuhaus, Zimmermann, Holler, and Zeller aim at predicting vulnerabilities [3]. Therefore, they make use of the modules' history, imports, and function calls. They build a knowledge base by the use of a VCS (CVS). The authors conclude that the occurence of vulnerabilities apparently correlates with the imported libraries.

A lot of work concerning dataflow analysis and incremental analysis is available in the field of compiler construction and compiler optimization. The results are partially applicable to security related static analysis. For example, Yur, Ryder, Landi, and Stocks provide an algorithm for an *Incremental Analysis of Side Effects for C* Programs [12]. Later on, Yur, Ryder, and Landi found a solution to the aliasing problem [11]. So, we can use their work to determine changes to variable values. This will lead us to possible security related faults. Ryder presents algorithms that implement incremental updates of a program's dataflow information [5]. Given a program and its dataflow graph (e.g. achieved by an exhaustive analysis), the algorithm takes a slightly modified program and outputs the nodes that are affected by the changes.

Wilander makes use of dependence graphs to visualise security properties of code [8]. These can be considered as blacklist and whitelist patterns to model bad and good parts of code respectively. In combination with incremental analysis, efficient security checks can be run on a temporary code base.

8. CONCLUSIONS AND FUTURE WORK

We have presented a server based approach for the integration of security mechanisms into the software development process. The advantages in comparison with the distributed host based approach have been shown. Furthermore, we have argued for the feasibility of our approach. As a next step, we will determine the pace of an implementation. Finally, we can conclude that the presented approach avoids ending up in the vicious circle of double burden, i.e. implementing new features and patching recent security flaws of released software before they are exploited.

That way, the chance to meet the next deadline is increased. The real timetable converges towards the planned one. Thus, there is more time for exhaustive testing in the end. The development process becomes more robust.

The set of security checkers applicable in the development phase is limited. The software might not be executable yet, so, for instance, fuzzing tools do not help. Additionally, no analysis tool guarantees for complete coverage. Thus, the quality of used tools is really important. Furthermore, the developers' acceptance of such a system is important for success. Those who feel controlled and distrusted use to compromise the system.

We are currently implementing a system that complies with the presented approach. It can serve as a basis for further advances. Important criteria include portability over

platforms, extensibility, interoperability with analysis tools and practical applicability. Furthermore, we are working on a good presentation of analysis results. Apparently, good results are worthless if the addressee (i.e. the developer) does not get the point.

Furthermore, we are collecting data which will allow us an empirical analysis of VCS usage (i.e. number of lines changes after last version, concurrent editing, conflicts, etc.), how changes in the software influence the analysis, and how patches and software are related.

Finally, we are running a broad test of client side analysis tools. These include free as well as commercial products. We will use the results as a basis for an evaluation of both approaches.

Acknowledgment

The author wishes to thank Daniel Schreckling for helpful conversations and comments on this work and some related ideas.

9. REFERENCES

[1] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[2] J. Merrill. Proceedings of the gcc developers summit. May 2003.

[3] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, New York, NY, USA, 2007. ACM.

[4] W. Royce. Managing the development of large software systems. In *Proc. IEEE Wescon*, pages 1–9, August 1970.

[5] B. G. Ryder. Incremental data flow analysis. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 167–176, New York, NY, USA, 1983. ACM.

[6] D. Schreckling and M. Johns. Automatisierter code-audit. *Datenschutz und Datensicherheit - DuD*, 31:888–893, December 2007.

[7] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[8] J. Wilander. Modeling and visualizing security properties of code. In *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden (SERPS'05)*, pages 65–74, Vasteras, Sweden, October 2005.

[9] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, pages 68–84, Karlstad, Sweden, November 2002.

[10] W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, University of Wisconsin - Madison, 1989.

[11] J.-S. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *International Conference on Software Engineering*, pages 442–451, 1999.

[12] J.-S. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for c software system. In *International Conference on Software Engineering*, pages 422–432, 1997.