Evaluating the Cost Reduction of Static Code Analysis for Software Security

Dejan Baca
School of Engineering,
Blekinge Institute of Technology
Sweden
dejan.baca@bth.se

Bengt Carlsson
School of Engineering,
Blekinge Institute of Technology
Sweden
bengt.carlsson@bth.se

Lars Lundberg
School of Engineering,
Blekinge Institute of Technology
Sweden
lars.lundberg@bth.se

Abstract

Automated static code analysis is an efficient technique to increase the quality of software during early development. This paper presents a case study in which mature software with known vulnerabilities is subjected to a static analysis tool. The value of the tool is estimated based on reported failures from customers. An average of 17% cost savings would have been possible if the static analysis tool was used. The tool also had a 30% success rate in detecting known vulnerabilities and at the same time found 59 new vulnerabilities in the three examined products.

Categories and Subject Descriptors D.2.8 [**Software Engineering]:** Metrics – product metrics; D.2.4 [**Software Engineering]:** Software/Program Verification; K.6.5 [**Management of Computer and Information Systems]:** Security and Protection.

General Terms Measurement, Security

Keywords *Security, Static code analysis, trouble report, early fault detection, code quality improvement, cost reduction, source code, false positive, Coverity Prevent*

1. Introduction

Flaws in software design and faults/bugs in software code are constantly being introduced into programs during their development. If certain conditions are present, these faults and flaws can propagate during runtime into failures that might be exploited as vulnerabilities [1], see also Figure 1. Failures and especially vulnerabilities increase the cost for the developers and require more time to be spent on maintenance instead of new features. Many developers rely on automated testing tools to verify their software and at the same time reduce development time [2]. Unfortunately, most of the testing is done to verify functionality and not to find vulnerabilities.

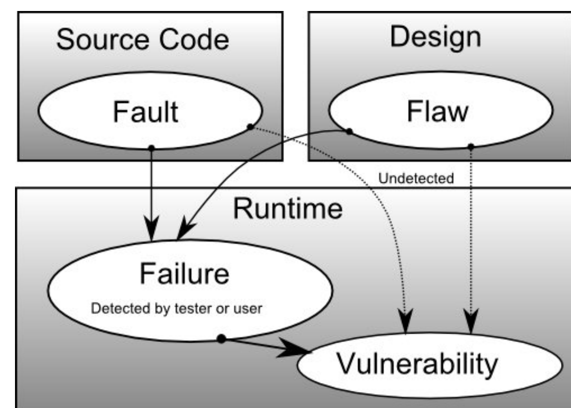

Figure 1. The relation between source code faults and design flaws resulting in visible failures that might propagate into known or unknown vulnerabilities.

Today, telecom developers are trying to decrease development time and shorten time to market. It is therefore of interest to detect and correct a fault before the software has been tested. The earlier an fault is detected the less it costs to correct the fault [14][16]. So, developers that want to lower development time and reduce the quantity of faults have to depend on early fault detection tools and methods; one of these is automatic static code analysis.

Automatic static analysis has been an area of early fault detection research for many years and has more recently been added to security engineering processes as an early tool for detecting vulnerabilities [3]. Researchers aim to minimize the human time and complexity needed during source code inspections, i.e. to automate source code reviews. The quest to automate and aid the source code review processes started with simple program checkers [4] followed by several commercial tools that for a variety of coding languages tried to achieve an automated source code review. These tools capture the most common faults in a particular

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS'08 June 7–13, 2008, Tucson, Arizona, USA.

Copyright © 2008 ACM 978-1-59593-936-4/08/06…$5.00.

coding language and help the developer to create more stable, reliable and secure code. The tools represent a varying degree of sophistication and some utilize complicated techniques such as abstract interpretation to achieve their goal of better automation. Different commercial vendors specialized their tools in areas such as security.

Several published papers have concluded that static code analysis tools (SAT) can detect code vulnerabilities [5]. However, the data are often laboratorial or based on assumptions that the reported warnings are indeed vulnerabilities. In this paper we have instead applied static analysis with reported vulnerabilities in large software systems and examined how effectively they are detected. In this case study we are trying to answer how effective static code analysis tools are in detecting vulnerabilities. These detected vulnerabilities present a cost saving opportunity for the developers. We also examine what is the correlation between SAT findings and reported vulnerabilities.

Section 2 presents previous studies that have examined static code analysis for vulnerability detection. Section 3 explains how the static analysis tool works and the development process used by the examined software, clarifying the need for early fault detection. Section 4 explains how the case study was performed and what data had been analyzed. In Section 5 all the results are presented. Section 6 discusses the effectiveness of the static analysis tool, the trouble report data mining and the possible cost and quality benefits of detecting the faults earlier.

2. Related work

There are several related articles examining the effectiveness of static code analyzers, both as a security tool and as a general fault finder. For most of these articles non commercial analysis tools are used or no industrial reference data are examined.

In Evans and Larochelle [6] a case study with the free tool Splint, the open source software wu-ftpd was used, where the discovered faults and false positive numbers were presented. They concluded that a static analysis tool can detect security faults but did not specify the content of the faults. Schuh [7] concluded that while today's code analysis tools are better it is still the expertise of the examiner that is the most important factor. He conducted the study on both C++ and Java code with the aid of three commercial checkers. The study used lab code with known vulnerabilities deliberately inserted, giving some clues about detected vulnerabilities. He also presented a high number of false positive from all of the investigated tools forming one of the most complete studies describing limitations with static code analysis. Carlsson and Baca [8] looked at mature telecom graded software, analyzed with the help of open source checkers, ITS4, Flawfinder and RATS. The study showed that a security static code analyzer could find security vulnerabilities but also created a large number of false positives. They did not compare their results with trouble reports or any maintenance cost with data from industry. The study also did a short survey into the return on investments associated with the work of using a static code analysis. Okun et al. [9] used the static code analysis tool Coverity Prevent and data from open source projects to determine if the projects using SAT have had any improvement in the number of vulnerabilities. Unfortunately, they did not produce any strong conclusions because there where too many interfering factors. This paper will try to answer the same question by investigating if already reported vulnerabilities could have been detected if SAT was used during development.

3. Research methodology

This section first explains how a fault was classified as a vulnerability. This makes it easy to determine what the most common effect of reported vulnerabilities is. A taxonomy to determine the cause of the vulnerabilities is then presented, followed by a process and software description and an in-detail explanation of the case study.

3.1 Taxonomy

We used the "Seven Pernicious Kingdoms" taxonomy from Katrina Tsipenyuk to group the vulnerabilities. This taxonomy focuses on implementation faults and is especially useful for static code analysis tools [10]. The taxonomy explains the cause of a vulnerability, but not necessarily the effect of it. It is divided into the following eight groups:

- Input validation and representation - Metacharacters, alternate encodings, and numeric representations cause input validation and representation problems.
- API abuse - An API is a contract between a caller and a receiver. The most common forms of API abuse occur when the caller fails to honor its end of the contract.
- Security features – Incorrect implementations or use of security features, e.g. incorrect encryption setup.
- Time and state - Distributed computation is about time and state, e.g., for more than one component to communicate, states must be shared, which takes time and therefore opens the door for race conditions.
- Errors - Errors are not only a great source of "too much information" from a program, they are also a source of inconsistent thinking that can be exploited.
- Code quality - Poor code quality leads to unpredictable behavior that often manifests itself as poor usability. For an attacker, bad quality provides an opportunity to stress the system in unexpected ways.
- Encapsulation - Encapsulation is about drawing strong boundaries around things and setting up barriers between them.
- Environment – Environment includes everything outside the code that is still critical to the security of the software.

3.2 Vulnerabilities

A failure is labeled as security vulnerable [11] if, under any conditions or circumstances, it results in denial of service, unauthorized disclosure, unauthorized destruction of data, or unauthorized modification of data. These represent the propagation or effect of an exploit.

- Denial of Service – Preventing the intended usage of the software. The most common Denial of Service attacks today are network based that exhaust system resources. Source code that does not always properly release a system resource can be exploited in the same manner, resulting in an exhaustion of resources.
- Unauthorized disclosure – Extracting data from the software that was meant to be secret, e.g. customer data or passwords. Most common attacks today are disclosing data from databases or web sites, often in the form of encrypted, or worse, plain text passwords.
- Unauthorized destruction of data – Destroying the data and preventing others from using it. Besides destroyed user data, configurations may be used to force the system to enter a default/unsafe state that later on can be exploited.
- Unauthorized modification of data – The data is not destroyed but instead altered to fit the need of the attackers. This is often the most serious result and often requires that the attacker gains full access to the system.

3.3 Development and SAT Process

The SAT Coverity Prevent was used during the case study. Prevent, derived from the Stanford Metal/xgcc research project, uses a combination of inter-procedural data flow analyses and statistical analysis to detect faults [12].

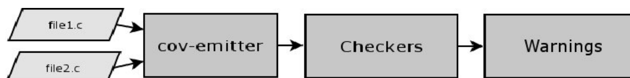

Figure 2. This figure explains the Prevent work flow [13].

After the tool has gathered the necessary data it utilizes checkers to detect faults. Each checker tries to match a specific category of potential faults. Figure 2 shows Coverity Prevents workflow were the tool first collects source code data and then relies on checkers to detect faults. The tool can also incorrectly claim that a fault is present (false positive).

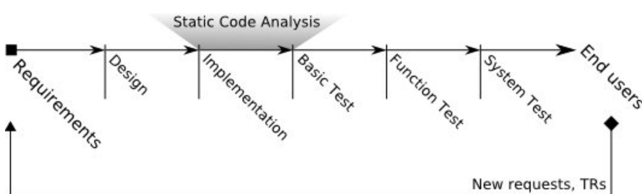

Figure 3. The intended usage of SAT in an example development cycle.

Figure 3 shows a simplified iteration of the products development process and the intended use of SAT. Because SATs are used early in development and are partially automated they can provide early fault detection. Two of the requirements for an early fault detection tool are fast execution time and the ability to run on non executable code. Because of these two requirements, dynamic checkers were excluded from the companies initial investigation where Coverity Prevent was deemed the best tool for the job. After the software has been completed it is released to customers. Any bugs that are found by the users are then reported back as trouble reports or bug reports (TR). A maintenance survey on the same development process suggests a 17 times increase in man hours to correct a TR compared to correct the fault during implementation [14]. Other studies have suggested up to a hundred fold increase in total cost [15].The higher cost supports the use of early fault detection tools to prevent a high number of TR’s.

3.4 Coverity Prevent checkers

The SAT is continuously improving and developing new checkers. At the time of the investigation these checkers stood out and were especially useful. Other studies have explained the tool and checkers more [17].

- NULL_RETURNS: A function that can return NULL must be checked before it is used. An attacker might be able to force the function to return an unexpected NULL and cause a segmentation fault.
- FORWARD_NULL: A program will normally crash when a NULL pointer is de-referenced. One situation this can happen is when the pointer has been checked against NULL and is de-referenced later. This check identifies such situation by checking all possible paths where such NULL dereferences can occur. The checker prevents denial of service attacks.
- REVERSE_NULL: Another situation this can happen is when the pointer is de-referenced before it has been checked against NULL. If the dereference is NULL, the check programmer should be warned to place the check against NULL before dereference.
- REVERSE_NEGATIVE: Sometimes a negative value is not advisable to use. One way to avoid such use is to check for negative value after a possible dangerous use. Attackers can effect loop iterations and copy operations if they can insert a negative values.
- SIZECHECK: Incorrect amount of memory allocation can lead to undetermined behavior and program crashes. By checking for inadequate memory allocations for a specific object it prevents memory out of bound errors.
- RESOURCE_LEAK: A memory leak can lead to program crashes. A leak of file descriptors, and socket can cause crashes and also have other harmful effects on the program. Apart from usual memory leak checks, it checks for interesting situations like aliasing as well.

- USE_AFTER_FREE: Heap values should not be used after they have been de-allocated, as it might lead to non-deterministic results when it is used. This also includes checks for double freeing of a pointer.
- UNINIT: The use of un-initialized variables can often result in nondeterministic behavior. Under some situations, it can also cause security vulnerabilities.
- OVERRUN_STATIC: This checker identifies invalid accesses to a static array, as it can cause buffer overruns that can ultimately lead to security vulnerabilities and program crashes.
- OVERRUN_DYNAMIC: Instead of examining static arrays this checker examines dynamic arrays. But the detected faults are of the same characteristic.
- NEGATIVE_RETURNS: If a value that is returned from a function can be negative and is used inappropriately, it can cause multiple errors such as memory corruption, crashes, infinite loops and so on.

4. Case study

The results are based on a case study that was performed on both mature telecom graded software and active open source projects. The software systems (A, B and C) are C++ based and are used as servers. The projects have different development processes. The case study was conducted on older versions of the product that had not used SAT's before. The source code also contained several already known vulnerabilities that were known through trouble reports. These vulnerabilities have been reported from outside the development team. The static analysis tool Coverity Prevent was chosen by a telecommunication and data communication systems manufacturer after an extensive internal investigation where several commercial and open source tools were compared. As a result of the internal investigation the tool Coverity Prevent is now deployed throughout the research and development units of the company. The newest version of the SAT was not used but instead the stable version from when the products were developed. This was necessary to ensure that our study will be as authentic as possible.

The products below have all passed several revisions and can be considered mature. Because of its size and age several developers have worked with the code in product A and B while in C a more dedicated small group of developers has been involved.

Product A had about 600.000 lines of analyzed code. The code base includes some third party code. The source code also includes a frame work which is included in the case study because TR's on the frame work are reported to the product. Product A receives, retrieves and stores data. It also handles and processes the user data. This is also the oldest product and has source code that was written several years ago.

Product B had about 300.000 lines of analyzed code and handles large amount of user data without doing any heavy processing. Its primary function is to shuffle data between different endpoints.

Product C had about 50.000 lines of analyzed code that serves and processes data. Compared to product A and B, it can not handle the same amount of possible input combinations. Product C mostly serves data to end users.

None of the examined versions of the products have had a formal security review as part of their ongoing development. All the product versions have been released and accumulated TR's for a minimum of one year.

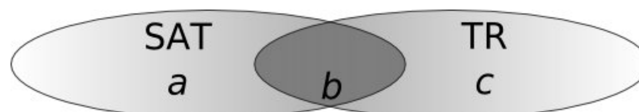

Figure 4. A fictive data collection with vulnerabilities from both the static analysis tool (SAT) and trouble reports (TR).

The static analysis tool (SAT) ran on an older already released version with reported faults. Therefore we had to analyze three variables of data, see Figure 4:

a = The correctly reported warnings from SAT not reported by TR's.

b = The intersection between SAT and the security warnings within TR, i.e. those TR warnings found by SAT.

c = The reported security TR not detected by SAT.

The dormant vulnerabilities (a) were either corrected in newer product versions or reported as TR after the case study. Any vulnerability that were dubious or hard to understand were also confirmed by testing them on the running product. These new faults present quality improvements done by the tool.

The intersecting faults (b) are possible cost improvements were the fault can be found earlier in the development process and therefore save time and money. These were found by examining what lines of code a TR had changed and then comparing if the SAT had issued a warning on these lines.

The known vulnerabilities (c) have all been reported, verified and corrected but were not detected by the SAT. If the TR results into changed code its origin can either been from a SAT detectable code fault or a more obscure design flaw.

4.1 Examining the tools output

The results from the SAT are divided into three groups.

False positives – These are any warnings that are incorrectly reported by the tool.

Security faults – This group represents warnings that are security related. It is also later on split up into two sub groups.

Functional faults – All the remaining correct warnings that did not fit into the either of the above groups were instead put here.

Security faults were split up into two sub groups. Vulnerabilities are warnings that could propagate into at least one of the four vulnerabilities classifications. The most contributing factor for the code quality group was rules and checks outside the source code that prevented exploitations. The source code was exploitable but the end product was not. These are therefore split into an own group, separate from the vulnerabilities and is not part of SAT (a) group in our model, Figure 4.

In the exploit tables in section 5, the total amount can be higher then the total amount of report warnings. This is possible because vulnerabilities sometimes can be exploited in more the one way.

4.1.1 Early TR detection effectiveness

The SATs effectiveness is depends on the tools capability to detect known vulnerabilities. Its effectiveness is easily calculated with the gathered data . i.e. an increased value indicates increased efficiency of the tool.

$$E = \frac{b}{b+c} \tag{1}$$

An E value of 1 shows that all TR are detected by SAT while 0 means none are found. E is easiest understood as percentage values, e.g. E = 0.2 means that the SAT detects 20% of the TR's.

4.1.2 Project cost improvement

To calculate the actual cost improvement we have to compare the cost of buying and using the tool divided by the cost the company paid to solve the TR's that the SAT also found, *b* in Figure 4.

The projects new TR cost with the SAT as a percentage compared to the old cost would then be

$$TR_{reduced} = \frac{Tool_{cost} + \left(\frac{Warnings}{Time_{factor}} \times Hour_{cost}\right) + \left(c \times TR_{cost}\right)}{TR_{cost}\left(b+c\right)} \tag{2}$$

$TR_{reduced}$ shows the percentage difference between the projects new maintenance cost and old maintenance cost.

TR_{cost} is the average total cost of a reported TR for that product. This value is the same for all three products and represents how much the companies development department on average has to spend per reported TR.

$Hour_{cost}$ represents the hourly developer cost and is constant for all three projects.

$Tool_{cost}$ the SAT has a, per lines of code, license. The products have different amount of code and therefore have different tool costs. But the license cost per line of code is the same.

$Time_{factor}$ is the number of warnings a developer can, on average, verify per hour. This value was determined from time reports and a follow up study. For all products 34 warnings can be examined per hour.

Warnings are the total of SAT warnings for the product, including false positives.

While *Warnings* and $Time_{factor}$ are published values, the other three have to be kept hidden due to company secrecy.

4.1.3 SAT Quality improvement

In this case quality is measured as the ability to prevent future TR's by detecting faults before testing, e.g. early fault detection. But to calculate quality we have to know the total amount of vulnerabilities.

To know the total amount of vulnerabilities (V_{tot}) all unknown vulnerabilities have to be counted also. Counting unknown vulnerabilities is impossible but we will try to estimate V_{tot} by comparing known vulnerabilities with new vulnerabilities that the SAT discovers. But SAT can only detect implementation vulnerabilities. Therefore all the design flaws have to be removed from the *c* group before calculating V_{tot}. Because we are using two different methods of detecting vulnerabilities it might be possible to use similar methods as capture, recapture [16] to determine what the total amount of vulnerabilities is.

By multiplying (a + b), the total number of vulnerabilities found by the tool, with b /(b + c) we may calculate V_{tot}:

$$V_{tot} = \frac{(a+b)(b+c)}{b} \tag{3}$$

5. Results

We divided the results between the three products and present them one by one. The first figure shows the SATs output while the first table presents the cause of the vulnerabilities and compare the SAT results with TR's. The second table shows the effect of the vulnerabilities where vulnerabilities sometime can be exploited in more than one way. Therefore the second table can have a higher total amount of vulnerabilities than the first table.

5.1 Product A

From the total amount of warnings 371 (22.1%) were false positive warnings and did not require any correction in the source code. 85 (5.1%) were classified by the author as security related and 48 (2.9%) were classified as vulnerabilities that could be exploited while the remaining were not exploitable but were bad code quality. The SAT found a new vulnerability every 12.500 lines of code. The product had 8 known vulnerabilities, 5 of them were implementation based.

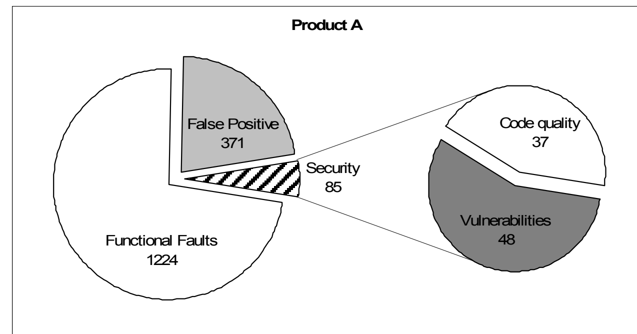

Figure 5. The total SAT results on product A, right circle explains security findings in more detail.

The SAT found all reported input validations. But surprisingly did not find all possible implementation vulnerabilities. Two TR's that were deemed as implementation faults were not detected by the SAT. The Time and state was most likely not detected do to missing checkers. The API abuse was very unclear. The tool had examined the file and the vulnerability was confirmed in a practical experiment. Also faults of a similar nature had been reported by the SAT before. The most likely conclusion is that the SAT misinterpreted the code and labeled the vulnerability as a false positive.

Table 1. Divided into the cause of the vulnerability, the finding of SAT or reports from TR's. Third column shows how many were detected by both methods.

Taxonomy	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Input validation and representation.	35	3	3
API abuse	1	1	0
Security features	0	1	0
Time and state	12	1	0
Errors	0	0	0
Code quality	37	-	-
Encapsulation	0	1	0
Environment	0	1	0

Due to the nature of the Coverity tool and its available checkers, the majority of the vulnerabilities are stack or heap specific, mostly buffer overflows. Only three vulnerabilities could not be used to repeatedly crash or lock up resources but they did instead present the possibility to leak information.

Table 2. The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way.

Failure	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Denial of Service	45	5	2
Unauthorized destruction	21	2	1
Unauthorized modification	23	3	1
Unauthorized disclosure	17	4	1

In product A the majority of vulnerabilities were not detected by TR's but instead newly found by the SAT. These were mostly local exploits and therefore unlikely to be reported as TR's.
Three reported vulnerabilities were detected by the SAT. Using eq. 1 the SAT had an E=37.5% TR detection effectiveness. The cost for the three vulnerabilities in the b group could have been reduced to a $TR_{reduced}$ =85% of the original total maintenance cost.

If the product would have used SAT and assuming the developers had spend the time to correct all the warnings, product A would have had 15% less in maintenance cost for security TR's and 45 unreported vulnerabilities would not have been present in the released product.

5.1.1 Product B

In product B only 5.3% of the total warnings were false positives. 12% of the warnings were security related witch is higher then Product A. But the number of vulnerabilities per lines of code is much smaller. The SAT tool found a new vulnerability every 42.850 lines of code. There were 7 known vulnerabilities, of these only 2 were implementation based.

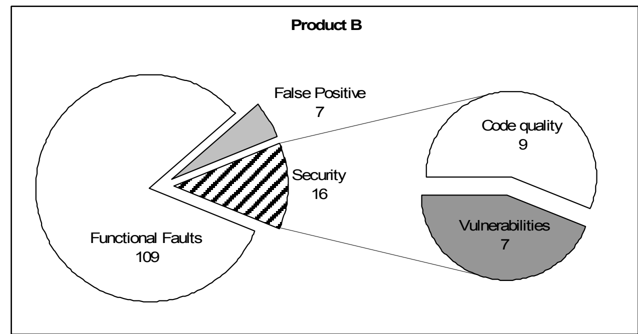

Figure 6. The total SAT results on product B, right circle explains security findings in more detail.

All the vulnerabilities were memory related and involved the writing or reading of Null values. An attacker could exploit these vulnerabilities with special crafted input messages. Some of the vulnerabilities required a special order of requests before the program entered an unsafe state. The majorities of TR's was design based and not do to poor implementation. Four design flaws were missing input validation and one flaw in the products API to outside programs.

Table 3. Divided into the cause of the vulnerability, the finding of SAT or reports from TR's. Third column shows how many were detected by both methods.

Taxonomy	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Input validation and representation.	3	5	1
API abuse	4	2	1
Security features	0	0	0
Time and state	0	0	0
Errors	0	0	0
Code quality	9	-	-
Encapsulation	0	0	0
Environment	0	0	0

Product B had only known DoS attacks and the SAT only found DoS vulnerabilities. This is probably because the product mostly does not process the data. There were seven reported TR's and the SAT found seven vulnerabilities. Two vulnerabilities were identified by both detection methods.

Table 4. The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way.

Failure	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Denial of Service	7	7	2
Unauthorized destruction	0	0	0
Unauthorized modification	0	0	0
Unauthorized disclosure	0	0	0

Product B did not process the data in any large amount and was therefore not receptive to a large variety of attacks. Only DoS attacks had been reported and all new vulnerabilities were also DoS attacks. But even the dormant unknown vulnerabilities were remotely exploitable.

All implementation faults were found by the SAT but none of the five design vulnerabilities. The SAT had an effectiveness of E=28.6% to detect vulnerabilities TR's. With eq. 2 the product would have had a new maintenance cost of $TR_{reduced}$ =83%, a 17% reduction. At the same time five undetected DoS vulnerabilities would also had been avoided.

5.1.2 Product C

Product C was the smallest of the products and had therefore a small development team. 2 (6%) of the warnings were false positive and 9 (27%) where security related. But a majority were just code quality issues and not exploitable. The SAT tool found a new vulnerability every 16.700 lines of code. The vulnerability detection rate was more similar to product A than B. The product had 8 known vulnerabilities, 2 of these were implementation based.

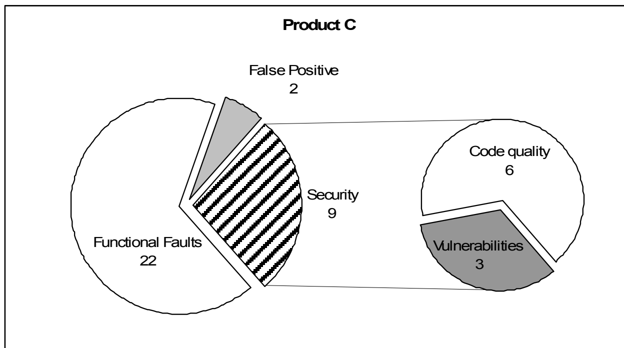

Figure 7. The total SAT results on product C, right circle explains security findings in more detail.

The SAT did not detect any other vulnerabilities than DoS attacks in this product. Because the product was relative small the number of detected vulnerabilities were also few.

In this product the SAT managed to detect incorrect encapsulation of data. The two worse TR's where design based with one lacking necessary input validation and the other allowing access by pass do to weak API.

Table 5. Divided into the cause of the vulnerability, the finding of SAT or reports from TR's. Third column shows how many were detected by both methods.

Taxonomy	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Input validation and representation.	1	5	1
API abuse	1	3	1
Security features	0	0	0
Time and state	0	0	0
Errors	0	0	0
Code quality	6	-	-
Encapsulation	1	0	0
Environment	0	0	0

Eight TR's had been reported and all of them could be used to launch DoS attacks. Two TR's where especially bad and could be exploit in all four ways. But the SAT did not find any new vulnerability except for one new DoS attack.

Table 6. The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way.

Failure	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Denial of Service	3	8	2
Unauthorized destruction	0	3	0
Unauthorized modification	0	2	0
Unauthorized disclosure	0	2	0

The SAT and had an effectiveness of E=25% in detecting TR's. The SAT also found one new vulnerability and provided a new $TR_{reduced}$ =77%. Showing this studies best cost reduction with 23%.

This product had the least benefit of the SAT in finding new or already known TR's, but at the same time saved the most money. The SAT also failed in detecting any of the non-DoS vulnerability. But it did detect all known implementation faults and at the same time one previously unknown. This product had the best saving potential because of its smaller size and therefore tool license cost, combined with a relative large amount of detected TR's compared to its size.

5.1.3 All products

Because the three products are different and have different development processes their results are not 100% compatible. But we combine them anyway to better show the similarities and differences between SAT and TR's.

The SAT managed to find some TR but did not locate all implementation faults even if it technically should be possible. The SAT was most effective at detecting Input validations. These are often memory related vulnerabilities. The SAT also found several new vulnerabilities that TR's had not. Only Input validations and API abuses were detected by both methods.

Table 7. Divided into the cause of the vulnerability, the finding of SAT or reports from TR's. Third column shows how many were detected by both methods. Summary of all vulnerabilities from the three products.

Taxonomy	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Input validation and representation.	39	13	5
API abuse	6	6	2
Security features	0	1	0
Time and state	12	1	0
Errors	0	0	0
Code quality	52	-	-
Encapsulation	2	1	0
Environment	0	1	0

In this case study the SAT had an average of 30.4% effectiveness in catching security TR's early. For the total project with all three products the SAT could have lowered the maintenance costs with 18% and at the same time detected 59 new vulnerabilities and prevented customers from having to report 7 security TR's.

Table 8. The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way. This table shows the combines results from all products.

Failure	SAT (a+b)	TR (b+c)	SAT∩TR (b)
Denial of Service	55	18	6
Unauthorized destruction	21	5	1
Unauthorized modification	23	5	1
Unauthorized disclosure	17	6	1

5.1.4 Code quality improvement

To calculate the possible quality improvement we have to estimate the total amount of exploitable code faults. We assume that the ratio between dormant vulnerabilities and implementation vulnerabilities that have both been reported and found by the SAT is constant. This assumption is based on the idea that for the SAT tool to detect the remaining implementation vulnerabilities its checkers needs to be improved or new added and would therefore find even more dormant vulnerabilities.

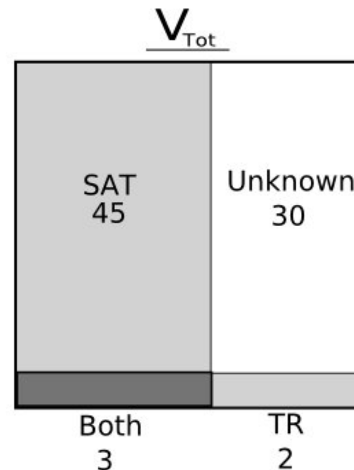

Figure 8. Total amount of implementations vulnerabilities in Product A. Using eq. 3.

In this case study there were still some known implementation based TR's that were not detected. One was relatively easy to understand and it was the lack of a concurrency checker that was responsible, newer version of the SAT has introduced concurrency checkers. But because the newer version was not released during the products development the vulnerability would not have been detected. On the second undetected implementation vulnerability it was more unclear. Similar faults had been reported before but this partial vulnerability was not. The most likely reason is that the tool decided that the fault was not significant enough and therefore did not report it. The SAT tried to minimize the false positive rate. Only Product A had undetected implementation TR's. If we assume that new or better checkers would find these and more vulnerabilities then according to Eq. 3 a new total amount of implementation vulnerabilities would have been 80 instead of the currently known 50. Product A had therefore only 60% of its possible code quality improvement, e.g. an improved SAT would have found even more vulnerabilities.

6. Discussion

When observing the results from the case study it is clear that in the particular case a static code analysis tool can detect faults that propagate into full vulnerabilities. The false positive rate of free/open source tools have been criticized in previous papers (see section 2 related works) and often discussed as a cause why developers do not use static code analysis. In this case study the false positive rate varied from 5-22%. Even the highest false positive rate is acceptable compared to previous studies.

The tool managed to find several vulnerabilities in mature, well tested and stable products. From these vulnerabilities the majority were stack or heap related. This shows that the tools security detection capabilities mostly rely on software's handling of memory, e.g. in the form of string based buffer overflows or memory problems due to tainting. 59 of the now known vulnerabilities were new vulnerabilities that the SAT found.

For this study the most important warnings were the intersecting (b) faults. They were both detected by SAT and reported as TR's. These faults have already cost the project money and time to verify and correct. Table 7 shows that 7 TR's were detected by the SAT (*b*). The TR's were either classified as Input validation or as API abuse. Both detection methods had found more categories but only these two had intersecting vulnerabilities. As for most of the vulnerabilities the effect was mostly DoS attacks. But there were two that could be exploited in more sinister ways.

Detecting design based failures is highly improbable, due to the data the tool uses, 16 (91%) of the undetected TR's were design based. But there were still 2 (9%) that should have been detected (in Table 1 there are two implementation vulnerabilities found by TR that SAT had not found), i.e. unlike design errors there were no technical restriction. Missing or faulty checkers is the most probable cause why the remaining implementation vulnerabilities were not found. The checker developer has to balance his checker in such a way that it does not provide too many false positives, as such some checkers would not be possible or hard to implement. So, the drawback of user friendly SATs is that some checkers do not catch all vulnerabilities of their type, e.g. the open source tool RATS may probably detect more vulnerabilities but with a much higher rate of false positives.

When comparing SAT and TR it is clear that their detection capability varies. SAT focuses mostly on memory handling and therefore detects mostly Input validations. TR's on the other hand detect vulnerabilities caused by both implementation and design. TR's from server software seem also to focus on remote exploits. But with a detection rate of about 30%, SAT still presents a partially automated method in detecting real vulnerabilities early. In the examined project with these three products, the maintenance cost for security TR's could have been lowered by 17% if a SAT would have been used. The SAT effectiveness and cost reduction are not in this study related to each other. Product A that had the best effectiveness result at the same time had the worse cost saving. Showing that the products size and complexity, assuming it increase the number of warnings, affect the cost of using the tool more then the added effectiveness saved money.

During this case study we have used TR's as a measurement for SAT effectiveness. As shown in the results TR's themselves are not a method that detects all vulnerabilities. But they do show vulnerabilities that have had a measurable cost and are therefore a good reference for cost saving actions.

7. Conclusion

We have looked at real-life trouble reports from three large software systems, consisting of approximately 1,000,000 lines of C++ code. There is a significant cost associated with handling the security related trouble reports in these systems. Our research study showed that, by using a static analysis tool a 17% cost reduction for reported security bugs would have been possible. One product even showed a 23% cost reduction. The cost reduction includes all costs associated with the SAT. Most of the vulnerabilities found were stack or heap related, i.e. the security detection capabilities mostly rely in the software's handling of memory. No design based vulnerabilities were detected and more implementation failures should have been possible to detect, i.e. when the static analysis tool is lowering the rate of false positives some vulnerabilities are dismissed. Almost 70% of the TR's were not found by Coverity, i.e. for these there were no reduction of costs. Product A had the best effectiveness with 37.5% of the TR detected while Product C had the worse with 25%, but because

The SAT also found dormant vulnerabilities that were not reported as TR's. A total of 2.6 times more vulnerabilities were detected by the SAT compared to the TR's. This means that static code analysis does not only reduce the cost. There is also a significant quality improvement due to the detection of dormant vulnerabilities.

References

[1] J. Viega and G. McGraw, Building Secure Software: How to Avoid Security Problems in the Right Way. Addison Wesley, 2002.

[2] A. Pretschner, W. Prenninger, S.Wagner, C. K¨uhnel, M. Baumgartner, B. Sostawa, R. Z¨olch, and T. Stauner. "One Evaluation of Model-Based Testing and its Automation." In Proc. 27th International Conference on Software Engineering (ICSE'05), 2005.

[3] S. Lipner, The Trustworthy Computing Security Develop-ment Lifecycle, Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), p.2-13, 2004

[4] S.C. Johnson., "Lint, a C program checker" Computer Science Tech. report 65, Bell Laboratories, 1978.

[5] B. Chess and G. McGraw. Static Analysis for Security. IEEE Security and Privacy, 2(6), 2004.

[6] D. Evans and D. Larochelle , "Improving security using extensible lightweight static analysis," IEEE Softw., vol. 19 (1) 42–51, 2002.

[7] J. Schuh and D. Stampley, CODE SCANNERS: FALSE Sense of Security?, Network Computing, 18, 7, ABI/INFORM Globalpg. 45, 2007

[8] B. Carlsson., and D. Baca, "Software Security Analysis - Managing Source Code Audit", 31st EUROMICRO Conference, 2005.

[9] V. Okun, W. F. Guthrie, R. Gaucher, P. E. Black, "Effect of static analysis tools on software security: preliminary investigation", Proceedings of Quality of Protection, p.1-5, 2007

[10] Tsipenyui, K., B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," presented at Automated Software Engineering, Long Beach, CA, 2005.

[11] C.E. Landwehr, ‘‘Formal Models for Computer Security,’’ ACM Computing Surveys 13(3), 247-278 1981.

[12] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2002.

[13] Coverity Prevent manual.

[14] L. Damm, "Faults-Slip-Through A Concept of Measuring the Efficiency of the Test Process", Journal of Software Process: Improving and Practice, Wiley InterScience, 11(1), pp. 47-59, 2006.

[15] Boehm, B. and Basili, V. Software Defect Reduction Top 10 List, IEEE Computer, Vol. 34, No. 1, January 2001.

[16] Briand, L.C., Emam, K. El, Freimut, B.G., Laitenberger, O., A Comprehensive Evaluation of Capture-Recapture Models for Estimating Software Defect content. IEEE Transactions on Software Engineering, 26:6, 518-540, 2000.

[17] Boehm B. 1983. Software Engineering Economics. PrenticeHall: Englewood Cliffs, NJ, USA.