Linköping Studies in Science and Technology

Dissertation No. 1503

Contributions to Specification, Implementation, and Execution of Secure Software

by

John Wilander

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2013

ISBN 978-91-7519-681-7
ISSN 0345–7524
Printed by LiU Tryck 2013

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-88330

This thesis is dedicated to my daughter
June Wilander.

Abstract

This thesis contributes to three research areas in software security, namely security requirements and intrusion prevention via static analysis and runtime detection.

We have investigated current practice in security requirements by doing a field study of eleven requirement specifications on IT systems. The conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions. A follow-up interview study addressed the reasons for the inconsistencies and the impact of poor security requirements. It shows that the projects had relied heavily on in-house security competence and that mature producers of software compensate for poor requirements in general but not in the case of security and privacy requirements specific to the customer domain.

Further, we have investigated the effectiveness of five publicly available static analysis tools for security. The test results show high rates of false positives for the tools building on lexical analysis and low rates of true positives for the tools building on syntactical and semantical analysis. As a first step toward a more effective and generic solution we propose decorated dependence graphs as a way of modeling and pattern matching security properties of code. The models can be used to characterize both good and bad programming practice as well as visually explain code properties to programmers. We have implemented a prototype tool that demonstrates how such models can be used to detect integer input validation flaws.

Finally, we investigated the effectiveness of publicly available tools for runtime prevention of buffer overflow attacks. Our initial comparison showed that the best tool as of 2003 was effective against only 50 % of the attacks and there were six attack forms which none of the tools could handle. A follow-up study includes the release of a buffer overflow testbed which covers 850 attack forms. Our evaluation results show that the most popular, publicly available countermeasures cannot prevent all of these buffer overflow attack forms.

This work has been supported by the National Graduate School in Computer Science (CUGS) commissioned by the Swedish government and the board of education.

Populärvetenskaplig beskrivning

Samhällen och människor har blivit beroende av datorer och mjukvara. I takt med att allt större värden och allt mer viktig information hanteras i persondatorer och på Internet så ökar risken för allvarlig IT-brottslighet. För tjugo år sedan spreds så kallade datorvirus med disketter och gjorde skada genom att radera eller förstöra information och programvara. Idag genomförs intrång via Internet i syfte att stjäla information, pengar eller att utkämpa cyberkrig.

Intrång i datorer sker ofta genom att utnyttja konstruktionsfel i programvara. Programmerare tänker inte alltid på att en illasinnad människa kan vilja "knäcka" hans eller hennes system. Ett vanligt exempel är så kallad *SQL-injektion*. Vi tänker oss ett program för inloggning som tar emot användarnamn och lösenord. Ett vanligt användarnamn skulle kunna vara "`johnwilander`". Om nu programmeraren inte på ett korrekt sätt tar hand om användarnamn som ser ut så här "`johnwilander' OR 1=1--`" så kan en elak användare radera hela databasen eller hämta all lagrad lösenordsinformation. Det kan se enkelt ut men just SQL-injektion är den vanligaste formen av intrång på Internet idag (se *Web-Hacking-Incident-Database* från organisationen WASC).

Sådana konstruktionsfel och utnyttjandet av dem ingår i ämnesområdet *mjukvarusäkerhet*. Den här doktorsavhandlingen ger bidrag till tre områden inom mjukvarusäkerhet:

1. Hur ställer man krav på bra mjukvarusäkerhet?
2. Kan vi hjälpa programmerare att upptäcka konstruktionsfel?
3. Kan vi göra datorprogram mer motståndskraftiga utan att behöva rätta till alla konstruktionsfel?

Vi har undersökt rådande praxis inom **kravställning av säkerhet** genom att granska elva kravspecifikationer **inom svensk offentlig upphandling** av IT-system. Vår slutsats var att **säkerhetskraven var undermåliga** och att det främst beror på tre saker: osammanhängande val av krav, osammanhängande detaljnivå i de krav som ställs och nästan inga krav på standardlösningar. Granskningen följdes upp med intervjuer där vi undersökte orsakerna till de undermåliga kraven och konsekvenserna för säkerheten. Bristfälliga krav berodde i stor utsträckning på att man som beställare förlitat sig på intern kompetens istället för att anlita experter på respektive område. Det visade sig dock att etablerade leverantörer av IT-system kompenserar för bristfälliga krav i allmänhet. Det ingår i en professionell leverans. Men någon sådan räddning verkar inte finnas när verksamheten har speciella krav på säkerhet och personlig integritet, till exempel inom sjukvård. Där måste verksamheten själv förklara och ställa krav för att slutprodukten ska få god säkerhet.

Vidare så har vi **utvärderat fem fritt tillgängliga verktyg för säkerhetsanalys av programkod**. Våra resultat visar på höga nivåer av falsklarm i verktyg som gör analys på så kallad lexikalisk nivå samt många missade säkerhetsproblem i verktyg som gör en djupare analys på så kallad syntaktisk eller semantisk nivå. **I ett steg mot ett bättre sätt att analysera programkod så föreslår vi så kallade dekorerade beroendegrafer**. Med sådana kan man både skapa modeller av säkerhetsegenskaper och sen söka i programkod för att se om egenskaperna finns eller saknas. Beroendegrafer kan representera både god och dålig programmeringspraxis. God praxis *måste* finnas med för att programkoden ska anses vara säker medan dålig praxis *inte* får finnas med. Det var viktigt att hitta ett sätt att representera både god och dålig praxis eftersom det finns närmast oändligt många sätt att *avvika* från god praxis och likaledes närmast oändligt många sätt att *undvika* dålig praxis. Vi har implementerat ett prototypverktyg där vi visar att beroendegrafer kan användas för att upptäcka bristande kontroll av inkommande heltal. Sådana brister har utnyttjats för att exempelvis köpa ett negativt antal varor (-10 bokhyllor eller -100 Ericssonaktier) och som resultat få en utbetalning.

Slutligen så har vi **utvärderat fritt tillgängliga verktyg för förhindrande av så kallade *buffer overflow*-attacker under drift**. Vår första studie visade att det bästa verktyget år 2003 bara förhindrade hälften av våra attackformer och att sex attackformer inte förhindrades av något verktyg alls. En uppföljande studie utvärderade verktyg tillgängliga år 2011 med hjälp av 850 varianter av buffer overflow-attacker. Även denna gång kunde vi visa att inte alla attackformer förhindras. Vår testbädd med 850 attackformer är släppt som fri mjukvara.

Acknowledgments

Finally, my PhD dissertation is written, our research papers are published, and an interesting chapter of my life has come to an end. I have a lot of people to thank for their support and encouragements.

First of all I'd like to thank my advisor, Professor Mariam Kamkar. Before I even applied for becoming a PhD student I asked some researchers I knew what was most important to consider. They all told me the same thing—your advisor will be most important, even more important than your choice of research topic. I can only agree. Imagine being the advisor of a student who leaves for industry before he's done, publishes a paper a year later and then publishes another paper four years further on. It takes a very considerate, calm, and professional advisor to bring such a PhD project to a closure. Mariam, you did. Thank you.

Second, I'd like to thank the National Graduate School in Computer Science (CUGS) commissioned by the Swedish government and the board of education. It financed most of our research. Thanks Anne Moe for your CUGS work.

Third, I'd like to thank a number of senior researchers who gave me in-depth feedback on my work. Thank you, Professor Kristian Sandahl, Professor Nahid Shahmehri, and Christoph Schuba.

Fourth, I'm very grateful for having been part of the Programming Environments Laboratory. I had a great time with you, fellow PhD students—Jon, Jens, Martin, Levon, Andreas, Kalle, Mattias, Anders, David, Robert, Peter A, Kaj, and Emma (R.I.P.). Thanks also to the senior researchers Professor Kristian Sandahl, Professor Peter Fritzon, Professor Christoph Kessler, and Professor Uwe Aßmann. And of course thanks to Bodil Mattsson-Kihlström, the lab's project administrator but also an important social piece in the PhD puzzle.

Fifth, I would like to thank all my undergraduate students, especially the ones taking my two courses in programming. Nowadays I meet you in industry and you seem to be doing great. At times I think of our lectures on static typing and laboratories on infix to postfix conversion with the Shunting-Yard Algorithm (*Järnvägsalgoritmen*). Always brings a smile to my face. The Computer Undergraduates Section nominated me as the best teacher two years in a row and I consider that my most valuable awards to date. And of course, I would never have succeeded in teaching if it wasn't for the support and inspiration from Jon Edvardsson, Jens Gustavsson, Anders Haraldsson, and Fredrik Kuivinen.

Sixth, I would like to thank all our co-authors—Jens Gustavsson, Pia Fåk, Nick Nikiforakis, and Yves Younan. Additionally I thank our paper previewers Crispin Cowan, David Byers, and Martin Johns, our Master's Thesis students Pia Fåk and Pontus Viking, our interviewees for the field study on security requirements, and all researchers who've granted us access to their tools. Together we made this possible. Thanks for the support!

Seventh, I'd like to thank David Byers and Viiveke Fåk for introducing me to computer security as an undergrad.

Eighth, I'm very thankful for the support I've had from my employers Omegapoint and Svenska Handelsbanken, as well as from the global community of OWASP (The Open Web Application Security Project).

Ninth, I'd like to thank my fellow musicians in the heavy metal cover band Superset who played a great part of my life during the Linköping years. We rocked! And to Rolle, Hånken, Bundy, Båkke, Martin, Rolemaster, Boni, Poseidon, Jellypope, and Scanner for filling those years with the uttermost fun, craziness and geekery. Music-wise I'm ever so thankful for the music of Esbjörn Svensson Trio, Running Wild, and Laserdance. It might seem like an odd mix but those three are the primary bands I've listened to during endless writing, compiling, and reading of memory dumps.

Last but not least I'd like to thank my family for their everlasting support. Thanks to my mother for teaching me "scientific" stuff as a child. Thanks to my uncle for teaching me engineering. Thanks to my grandmother for spoiling me. Thanks Rama for telling me to "Go get your PhD" and thanks Johanna and June for the joy you bring to my everyday life.

John Wilander
Stockholm, February 7th, 2013

Contents

Chapter 1

Introduction

"To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products."

—Edsger W.Dijkstra, *The Humble Programmer* [1]

1.1 Background and Motivation

Computer software products are among the most complex artifacts, if not *the* most complex artifacts mankind has created (see Dijkstra's quote above). Securing those artifacts against intelligent attackers who try to exploit flaws in software design and construct is a great challenge too.

This thesis contributes to the research field of *software security.* Software as an artifact meant to interact with its environment including humans. Security in the sense of withstanding active intrusion attempts against benign software.

1.1.1 Software Vulnerabilities

Software can be intentionally malicious such as malicious *viruses* (programs that replicate and spread from one computer to another and cause harm to infected ones), *trojans* (malicious programs that masquerade as benign) and software containing *logic bombs* (malicious functions set off when specified conditions are met).

However, attacks against computer systems are not limited to intentionally malicious software. Benign software can contain *vulnerabilities* and such vulnerabilities can be exploited to make the benign software do malicious things. A successful exploit is often called an *intrusion.*

Vulnerabilities can be responsibly reported by first creating a so called CVE Identifier—a unique, common identifier for a publicly known information security vulnerability [2]. Identifiers are created by CVE Numbering Authorities for acknowledged vulnerabilities. Larger software vendors typically handle identifiers for their own products. Some of these participating vendors are Apple, Oracle, Ubuntu Linux, Microsoft, Google, and IBM [3].

The National Institute of Standards and Technology (NIST) has a statistical database over reported software vulnerabilities with a publicly accessible search interface [4]. Two specific types of vulnerabilities are of specific interest in the context of this thesis, namely buffer overflows and format string vulnerabilities in software written in the programming language C. The statistics for Buffer Errors and Format String Vulnerabilities are shown in Figure 1.1 and Figure 1.2.

Reported software vulnerabilities due to buffer errors have increased significantly since 2002. Their percentage of the total number of reported vulnerabilities has also increased from 1–4 % between 2002 and 2006 to 10–16 % between 2008 and 2012 [4]. These statistics are in stark contrast to the statistics from CERT that Wagner et al used to show that buffer overflows represented 50 % of all reported vulnerabilities in 1999 [5]. We have not investigated if there are significant differences in how the two statistics were produced. Still, up to 16 % of all reported vulnerabilities is a significant number.

The reported format string vulnerabilities peaked between 2007 and 2009 but have never reached 0.5 % of the total [4]. Our experience is that format string vulnerabilities are less prevalent, easier to fix, and harder

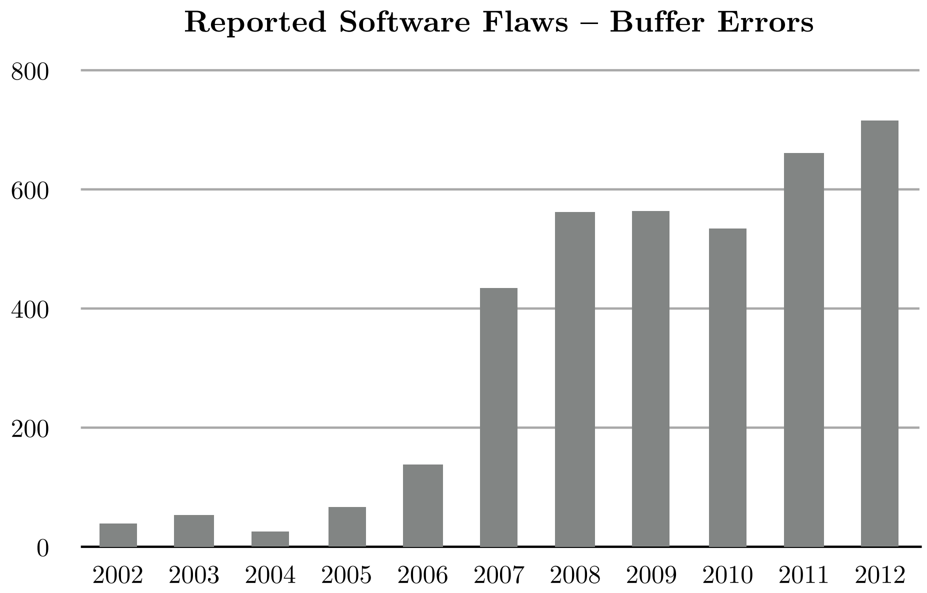


Figure 1.1: **Buffer Errors 2002–2012** according to vulnerability statistics from the NIST National Vulnerability Database.

to exploit than buffer overflow vulnerabilities. Nevertheless format string vulnerabilities are still being used for exploitation such as the Corona iOS Jailbreak Tool [6].

1.1.2 Avoiding Software Intrusions

Intrusion attempts or attacks are made by malicious users or *attackers* against *victims*. A victim can be either a machine holding valuable assets or another human computer user. Securing software against intrusions calls for anti-intrusion techniques as defined by Halme and Bauer [7]. We have taken the liberty of adapting and reproducing Halme and Bauer's figure showing *anti-intrusion approaches* (see Figure 1.3).

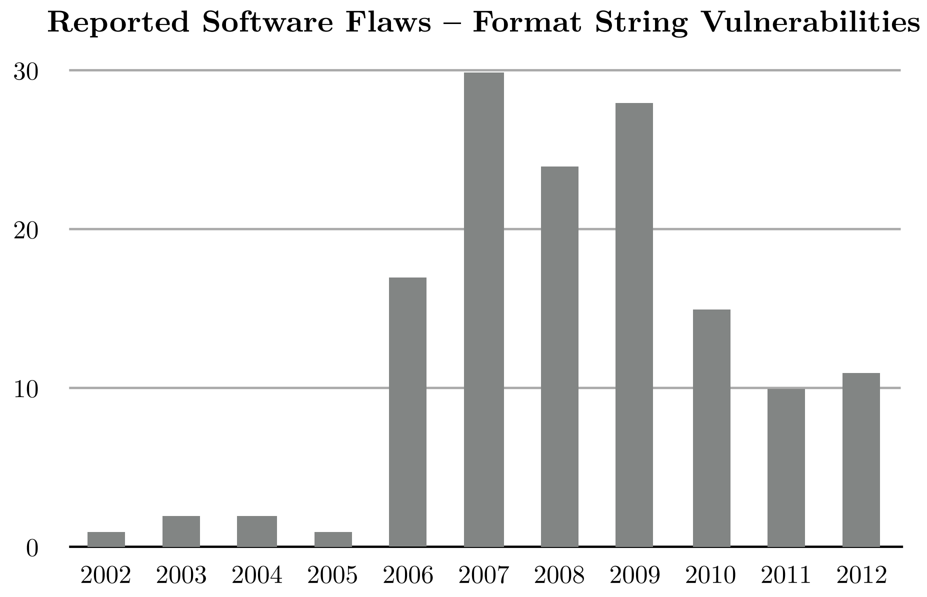


Figure 1.2: **Format string vulnerabilities 2002–2012** according to vulnerability statistics from the NIST National Vulnerability Database.

Preempt —strike offensively against likely threat agents prior to an intrusion attempt. May affect innocents.

Prevention —severely handicap the likelihood of a particular intrusion's success. In the context of this thesis prevention involves software with protection built-in and pre-release reports to programmers about likely vulnerabilities.

Deter —increase the necessary effort for an intrusion to succeed, increase the risk associated with an attempt, and/or devalue the perceived gain that would come with success.

Deflect —leads an intruder to believe that he or she has succeeded in an intrusion attempt, whereas in fact the intrusion was redirected to

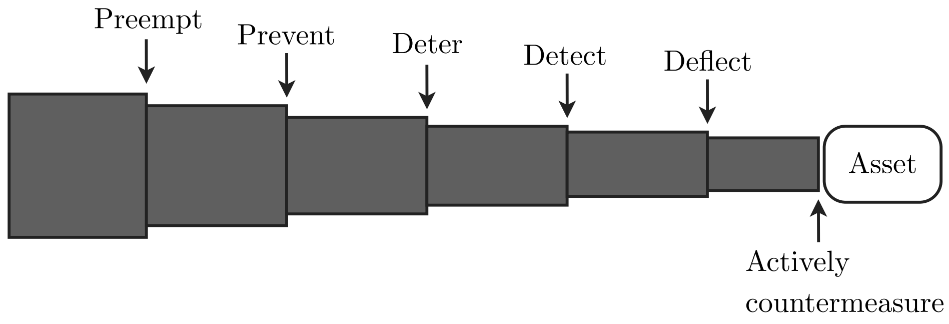


Figure 1.3: **Anti-Intrusion Approaches**. Intrusions can be stopped in at least six ways—preemption, prevention, deterrence, deflection, detection, and by active countermeasures as the intrusion attempt is carried out. The figure is a slightly adapted version from Halme and Bauer's anti-intrusion techniques.

where harm is minimized.

Detect —discriminate intrusion attempts and intrusion preparation from normal activity and alert the operations. Detection can also be done in a post mortem analysis.

Actively countermeasure —counter an intrusion as it is being attempted.

1.2 Research Objectives

There are many ways to achieve more secure software. Microsoft's Security Development Lifecycle (SDL) defines seven phases where security enhancing activities and technologies apply [8]:

1. Training
2. Requirements

3. Design
4. Implementation
5. Verification
6. Release
7. Response

Further things can be done in an even wider scope. Programming languages can be constructed with security primitives which allow programmers to express security properties of the system they are writing—so called security-typed languages, a part of language-based security [9, 10]. Operating systems and deployment platforms can be hardened and secured both in construction and configuration.

Our research objectives have been on the Requirements and Implementation phases of Microsoft's SDL and on hardening of the runtime environment for software applications.

1.2.1 Eliciting and Specifying Security Requirements

A software product owner or an organization purchasing software needs to convey any security requirements they have to the producer of the software. Functional security requirements such as authentication and authorization are well-known to experienced software users and are thus likely to be specified in the software requirements, at least on a high level.

However, non-functional security requirements such as the absence of known security vulnerability types or the properties of a certain encryption algorithm are not visible to software users nor are they part of general IT knowledge. Therefore such security requirements will likely not be specified by a product owner or purchaser of software. In the case of a successful attack the product owner might express that non-functional security requirements were implicit. The producer in turn might respond that security measures—functional as well as non-functional—cost time and money, and that the product owner has to be explicit if such time and money is to be spent.

1.2.2 Implementation

Software development in general is hard. Developing reasonably secure software is even harder. Programmers need training as well as a proper toolbox to tackle all the challenges in software development—for instance performance, maintainability, scalability, availability, and security.

Security vulnerabilities in software often boil down to implementation flaws. For example side effects are ignored or unrecognized, APIs are used in unintended ways, user input is not properly validated against the right data model, or data is used without proper context adjustments.

Programming tools such as integrated development environments, continuous integration servers, static analysis, and API defaults all have to help developers implement more secure systems.

1.2.3 Hardening the Runtime Environment

Security vulnerabilities in software will keep escaping even the best of developers and security-oriented tools. One of the reasons is legacy software—most software reliant organizations have a significant amount of software developed before certain vulnerability types were known and preventive tools were in place. Another reason is the evolving knowledge of how software can be exploited. A third reason is the general impossibility of bug free software.

Therefore we need to have hardened and monitored runtime environments. Software in deployment can be attacked. If an attack occurs the software should try to protect itself, for instance by doing integrity checks on its state and terminating execution if integrity violations are found. Further, software in deployment needs to be monitored for abnormal or malicious behavior.

1.2.4 Problems Addressed by This Thesis

How are Secure Software Requirements Currently Specified?

We need more information on how industry handles security requirements today to be able to move forward in that part of the security development lifecycle. Are stakeholders such as product owners and project leaders aware of security? Do they focus on functional as well as non-functional security requirements in their elicitation processes? If there are deficiencies, how do they impact the security of implemented systems?

How Can Static Analysis Help Developers Implement Secure Software?

Compile-time developer feedback from static analysis tools have many benefits. The analysis can be automated, it does not require a dedicated testing environment, it does not need complex test data generation to be able to analyse the complete code base, and problem reports can be mapped to exact lines of code.

Several static analysis tools for security have been developed both in academia and industry. How effective are they in finding real security vulnerabilities? How usable is their output to developers? Can they be made to report on both presence of bad programming practice and absence of good programming practice?

Can Runtime Protection Solve the Buffer Overflow Problem?

Buffer overflows have plagued C/C++ software for decades. It is an important field of software security. Several protection mechanisms have been presented and implemented in both academia and industry. How effective are they against the various kinds of buffer overflow attacks? Can we measure them in a repeatable way?

1.3 Contributions and Overview of Papers

The following sections summarizes of our research contributions categorized into Specification, Implementation, and Execution of secure software. The contributions are also clearly connected to their respective papers.

1.3.1 Specification of Secure Software

Our contributions to specification of secure software are in empirical field studies of requirements engineering practice for security.

Field-Study of Practice in Security Requirements Engineering

In 2005 and 2007 we published two closely related papers on industry practice in security requirements engineering. The first of these papers presented a field study of eleven software projects including e-business, health care and military applications. We categorized the security requirements as functional and non-functional and found that 76 % of the security requirements are functional despite security being a popular example of non-functional aspects of software.

The overall conclusion was that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions. This work was done jointly with Jens Gustavsson and is presented in Paper A.

Interview Study on the Impact of Security Requirements Engineering Practice

As mentioned above we published two closely related papers in 2005 and 2007. The second of these papers addressed two important questions which remained open since the first study; what are the reasons for the requirements inconsistencies, and what is the impact of such poor security requirements?

We performed in-depth interviews with three of the customers from the previous study. The interviews showed that mature producers of software (in this case IBM, Cap Gemini, and WM-Data) fulfill unspecified but reasonable requirements in areas within their expertise, namely software engineering. An example of this kind of over-delivery was found to be software maintainability requirements. But in the case of unspecified security and privacy requirements specific to the customer domain, such over-delivery was not found. In all three cases the neglect or underspecification of domain-specific requirements had led to security and/or privacy flaws in the systems. Our conclusion is that special focus needs to be put on domain-specific security and privacy needs when eliciting customer requirements. This is also joint work with Jens Gustavsson and it is presented in Paper B.

1.3.2 Implementation of Secure Software

Our contributions to implementation of secure software are in the area of compile-time analysis of source code and reporting and visualizing potential security vulnerabilities back to the programmer.

Static Analysis Testbed and Tool Evaluation

In 2002 we published a static testbed of 44 function calls in C implementing safe and unsafe testcases for buffer overflow and format string vulnerabilities. The testbed was used to empirically compare five publicly available tools for static analysis. We believe this to be "the first systematic benchmarking study concerning static analysis for security" as stated by Johns and Jodeit [11]. The work is presented in Paper C.

Modeling and Visualizing Security Properties of Code

In 2005 we published two closely related papers on a formalism for modeling, visualizing, and pattern matching security properties of code. This section covers our contributions from the first of those papers.

The paper discusses modeling security properties, including what we call the *dual modeling problem.* Security vulnerabilities can manifest themselves as presence of bad programming practice or absence of good programming practice. Thus, when reasoning about security properties of code we need to model both. As an example we show 1) a model of correct input validation where its *absence* implies a potential security vulnerability, and 2) a model of incorrect multiple freeing of the same memory where its *presence* implies a potential security vulnerability.

We propose dependence graphs decorated with type and range information as a generic way of modeling security properties of code. These models can be used to characterize both good and bad programming practice.

Continuing, we exploit the absence of good programming practice to produce potentially infinite models of bad programming practice. These model variations can be used to rank the severity of potential vulnerabilities. This work is presented in Paper D.

Pattern Matching Security Properties of Code

As mentioned above, in 2005 we published two closely related papers on a formalism for modeling, visualizing, and pattern matching security properties of code. This section covers our contributions from the second of those papers.

The paper reports on our proof of concept implementation of pattern matching security properties of code using dependence graphs. The graph models of the programs were built with Grammatech's tool *CodeSurfer* [12]. The tool is called *GraphMatch* and it can detect integer input validation flaws.

GraphMatch performed well on our synthesized micro benchmarks whereas real-life applications posed a harder problem. We checked wu-ftpd 2.6-4 which consists of approximately 20 KLOCs and produces a dependency graph with approximately 130,000 vertices. An analysis for integer input validation flaws took 15 hours on a 2.66 GHz Pentium 4. GraphMatch produced three warnings, two false positives and one true positive. The implementation work was done by Pia Fåk and published as her Master's thesis, supervised by Wilander and Kamkar [13]. The GraphMatch work is presented in Paper E.

1.3.3 Execution of Secure Software

Our contributions to the execution of secure software are in the area of runtime buffer overflow prevention.

Runtime Buffer Overflow Prevention Testbed and Tool Evaluation

In 2003 we published a runtime testbed of 20 working buffer overflow attacks. The testbed was used to empirically compare four publicly available tools for runtime intrusion prevention and showed that the best tool was effective against only 50 % of the attacks and that there were six attack forms which none of the tools could handle.

We believe this to be the first systematic benchmarking study concerning runtime analysis for security since earlier studies either did not do testing at all or did not take a structured approach (see Related Work in Section 6). This testbed has been used to demonstrate subsequent progress in the field [14, 15, 16, 17, 18, 19] and the outcome of our evaluation was used to motivate further preventive research [20, 21, 22, 23]. Microsoft Research ported the testbed to Windows for internal purposes and Silberman and Johnson presented the testbed at Black Hat USA 2004 [24]. This work is presented in Paper F.

In 2011 we revisited the topic with a new runtime testbed of 850 working buffer overflows named RIPE, Runtime Intrusion Prevention Evaluator. It was released as free software and we used it to evaluate more recent protection tools and techniques such as ProPolice, LibsafePlus+TIED, CRED, and Ubuntu 9.10 with non-executable memory and stack protection. The RIPE study was joint work with Nick Nikiforakis and Yves Younan at Katholieke Universiteit Leuven. A previous version of RIPE was implemented by Pontus Viking and published as his Master's thesis, supervised by Wilander and Kamkar [25]. The RIPE work is presented in Paper G.

1.4 List of Publications

This thesis comprises the following published, peer-reviewed[1] papers.

A Comparison of Publicly Available Tools for Static Intrusion Prevention by John Wilander and Mariam Kamkar. In the Proceedings of the 7th Nordic Workshop on Secure IT Systems (Nordsec 2002), November 7-8, 2002, in Karlstad, Sweden.

A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention by John Wilander and Mariam Kamkar. In the Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03), February 5-7, 2003, in San Diego, California.

Security Requirements—A Field Study of Current Practice by John Wilander and Jens Gustavsson. In the E-Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS 2005), August 29, 2005, in Paris, France.

Modeling and Visualizing Security Properties of Code using Dependence Graphs by John Wilander. In the Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden (SERPS'05), October 20-21, 2005, in Västerås, Sweden.

Pattern Matching Security Properties of Code using Dependence Graphs by John Wilander and Pia Fåk. In the Proceedings of the 1st International Workshop on Code Based Software Security Assessments (CoBaSSA 2005), November 7, 2005, in Pittsburgh, Pennsylvania, USA.

The Impact of Neglecting Domain-Specific Security and Privacy Requirements by John Wilander and Jens Gustavsson. In the Proceedings of the 12th Nordic Workshop on Secure IT Systems (Nordsec 2007),

[1]SERPS is a national conference on software engineering research and CoBaSSA is a workshop for early work. Our two papers published there were indeed peer-reviewed but with a high acceptance rate. The acceptance rates for the other conferences were all 25 % or below.

October 11-12, 2007, in Reykjavík, Iceland.

RIPE: Runtime Intrusion Prevention Evaluator by John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar and Wouter Joosen. In the Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011), December 5-9, 2011, in Orlando, Florida.

Chapter 2

Research Methodology

We have used four methods in our research—document inspection, qualitative interviews with transcription, synthesized micro benchmarking, and proof of concept verification.

2.1 Document Inspection

We chose to do document inspection in our field study of current practice of security requirements engineering. Requirement specifications used for public procurement by the Swedish Government or local authorities are public documents. We inspected eleven requirements specifications of IT systems being built 2003 through 2005 trying to find all instances of security requirements. The inspection consisted of both a manual read and computer-aided search for keywords.

The main reason for basing our study on document inspection was availability. Retrieving the documents required no specific permissions, negotiations, or agreements. Making contact with each project asking for further material might have skewed the results for certain projects compared to projects where we did not get further information.

2.1.1 Limitations

First, we only had access to the specifications in formal, written form. If they were in fact augmented by explanations, meetings, and email conversation we did not cover that in our analysis. However, public procurement is a formal process and requirements specifications should be complete to allow for fair competition among bidders. In our subsequent interview studies we did not get the impression that our analysis of the written specifications gave an incomplete picture. Refinements had been done but only after a supplier had been appointed.

Second, the choice of keywords to look for was limited by our experience and knowledge in the fields of security and privacy. Although we took a broad approach (for instance including logging in general as a security requirement) we may have missed security requirements simply because we didn't understand that certain requirements were related to security.

Finally, we did not make use of any formal process for document inspection such as Fagan inspection [26]. While a formal inspection process would have given rigor to our work we were not inspecting the documents for flaws, rather browsing for security and privacy requirements with a broad perspective.

2.2 Qualitative Interviews with Transcription

In our study "The Impact of Neglecting Domain-Specific Security and Privacy Requirements" (Paper B) we carried out three qualitative interviews with customer project leaders. The goals were to investigate the impact of the requirements deficiencies we found in the previous study (Paper A) and verify the hypotheses we had of their causes. The interviews were *semi-structured* with a pre-defined set of questions but allowing for new questions to be brought up during the interview [27]. Interviewees had full freedom to formulate their answers. The interviews were recorded and later transcribed verbatim to allow for analysis, including cross-referencing for consistency.

2.2.1 Limitations

The first and foremost limitation of our interviews is their number. With only three interviews you cannot draw general conclusions. However, our goal was to test our hypotheses from the previous study which used document inspection. By picking the top three from the previous study in terms of security requirements quality we set an upper bound on our analysis, i.e. the other systems were unlikely to show substantially better results when verified against our hypotheses.

Second, structured interviews [27] with exactly the same questions in the same order would have allowed for a detailed comparison between the three interviews. We opted out of a structured approach out of three reasons:

- A detailed comparison would not allow us to draw general conclusions given we only carried out three interviews. Not even interviews with all eleven projects from the previous study would have allowed for a proper comparison since the projects were so diverse in scope, size, and requirements quality. Furthermore, we had a hard time scheduling even the three since the project managers were busy.

- We did not know the knowledge level of the interviewees in advance which means it would have been risky to decide on a given level of abstraction and detail. In the worst case interview one and two would have gone well and then the third would not have provided any valuable answers.

- We wanted to ask questions on specific security requirements specified by each project and the difference in the three projects would not have allowed a question-by-question comparison except for a set of general questions.

In hindsight a combined approach would probably have been better—one fully structured part and one semi-structured part. The structured part could have focused on questions that can be compared between projects such as “Have you had security incidents since your first release? If so, how many?” and “Would you consider security and privacy requirements simple, fairly simple, hard, or very hard to specify?”.

Third, there is a possibility that our interviewees were negatively affected by the recording. We cannot know if they would have answered our questions differently with out being recorded. This issue was discussed in advance and we concluded that the possibility to transcribe all interviews would allow for a more careful analysis as opposed to just taking notes, and that this outweighed the potential drawbacks. We did not want to record in secrecy since all the terms had to be clear for the interviewees to accept the publishing of the results.

Finally, all quotes in the paper have been translated into English since the interviews and transcriptions were carried out in Swedish. Nuances and details always run the risk of being lost in translation, especially since it was carried out by the authors, not professional translators. However, our analysis of the interviews was all done in Swedish and only the last step, quoting the interviewees was translated.

2.3 Synthesized Micro Benchmarking

In all three of our comparative studies of intrusion prevention tools (Papers C, F, and G) we've used synthesized micro benchmarking suites built from small, deliberately vulnerable snippets of code. Others have used real-world benchmarks (see Related Work, Section 3.1.1) and a third option is to use educational applications [11].

The benefits of using micro benchmarks such as ours have been discussed by Johns and Jodeit [11]. First, they can be fine-grained meaning that even detailed differences can be evaluated. Second, they allow for full coverage of vulnerability classes, for instance via combinatorial space exploration. Third, tests that fail for a certain tool can easily be modified to investigate the cause of the failure. Fourth, micro benchmarks provide a controlled environment where researchers have a high confidence in the real number of vulnerabilities present as opposed to real-life code with a few *published* vulnerabilities but no knowledge of the real number. Finally, the relative small size of micro benchmark suites makes analysis of test results much more feasible.

2.3.1 Limitations

The drawbacks of micro benchmarks have been explained in several of the surveys covered in Related Work, Section 3.1. First, micro benchmarks do not test the tools' abilities to handle real-life code and real-life code's complexities such as build scripts, meta programming, and linking to non-standard libraries. Second, they do not test the scalability and usefulness of the tools on real-life code. Finally, they do not give convincing answers as to whether a certain tool would have found vulnerabilities that are known to have been in production.

2.4 Proof of Concept Verification

We used a proof of concept implementation to verify the our proposed formalism for pattern matching security properties of code—System Dependency Graphs decorated with type conversion information. While only pattern matching for one type of vulnerability the implementation gave us hands-on experience on usefulness and scalability of a straight-forward implementation.

The main reasons we did a proof of concept verification were to a) investigate the feasibility of the analysis in terms of execution time, and b) testing the relevance of our input validation model by checking the exploitability of any model mismatches we found in real-life code.

2.4.1 Limitations

First, a common limitation of proof of concept verifications, also applicable to our study, is the lack of availability to the research community. All too often the source code and build scripts are kept secret. This was also the case of our GraphMatch tool. In hindsight it would have served the research community much better to release the code. But at the time we had planned to continue working on the tool. We could release it today but that would require us making sure it builds and runs on a currently available system. Our results could have been reproduced and verified if we had published the code.

Second, our tool most likely contains bugs that affected the outcome of our verification. Perhaps the system under analysis (Sendmail) contained several security bugs of the type GraphMatch was looking for but they remained undetected because of a bug of our own. This could have been investigated with a synthesized micro benchmark such as the ones described in Section 2.3.

Finally, our proof of concept verification was not compared to other approaches in terms of effectiveness or efficiency. At the time, we were not aware of any publicly available tool trying to solve the same problem. However, we could have done such comparison ourselves given the results from GraphMatch, i.e. we could have investigated which other analysis methods could have found the same bugs GraphMatch found, only more effectively and/or efficiently.

Chapter 3

Related Work

Each of the papers included in this thesis includes references to previous work related to the problems addressed in that particular paper. This presentation of related work includes more recent work in the areas of compile-time intrusion prevention and security requirements engineering where a lot of research has been done since our most recent publications in 2005 and 2007 respectively.

3.1 Compile-Time Intrusion Prevention

Compile-time intrusion prevention tools try to prevent attacks by finding security vulnerabilities in the source code so that programmers can remove them. Removing all security bugs from a program is considered infeasible which makes the compile-time solution incomplete [28]. The two main drawbacks of this approach are that someone has to keep an updated database over programming flaws or best practice to analyze or check for, and since the tools only *detect* vulnerabilities the user has to fix the problem.

3.1.1 Static Analysis

Several steps forward have been taken since our comparative study of static analysis tools for security in 2002.

Static analysis for security has become an established business with companies such as HP (formerly Fortify), Coverity, IBM (formerly Ounce Labs), Veracode, and GrammaTech. The business term is *Static Application Security Testing*, SAST. A fairly complete collection of available SAST tools can be found on the NIST web page for Source Code Security Analyzers [29].

Many of the recent static analysis studies and techniques have been targeted towards mainstream object-oriented languages such as Java and C#, and web applications including languages like PHP and JavaScript. However, this section is limited to static analysis of C, for the purpose of finding buffer overflow and format string attacks or for the purpose of evaluating existing tools. The limitation is due to the scope of this thesis.

NIST's Static Analysis Tool Exposition

The National Institute of Standards and Technology (NIST) has published three comparative studies on static analysis tools for security called Static Analysis Tool Exposition, *SATE*. The most recent one at the time of writing this thesis was carried out in 2010 and published in 2011, called SATE 2010 [30].

SATE 2010 covers a C/C++ track and a Java track. SATE 2010 used a set of programs and among them a set of *CVE-selected test cases* from the CVE database [31] (CVE stands for *Common Vulnerabilities and Exposures*). The CVE-selected test cases were pairs of programs: an older, vulnerable version with publicly reported vulnerabilities (CVEs) and a fixed version, that is, a newer version where some or all of the CVEs were fixed. For the CVE-selected test cases, they focused on tool warnings that corresponded with the CVEs.

The C/C++ track covered three systems—Dovecot secure IMAP and POP3 server ($\approx$200 KLOCS), Wireshark network protocol analyzer ($\approx$1,600 KLOCS), and Google Chrome web browser ($\approx$4,000 KLOCS). For C/C++ the following static analysis tools participated; Concordia University MARFCAT, Coverity Static Analysis for C/C++, Cppcheck, Grammatech CodeSonar, LDRA Testbed, Red Lizard Software Goanna, Seoul National University Sparrow, and Veracode SecurityReview.

Selected subsets of tool reports were analyzed and compared. Three types of selection were done—Random, Related to manual findings by experts, and Related to CVEs.

Correctness of reports were categorized in True security weakness, True quality weakness, True but insignificant weakness, Weakness status unknown, and Not a weakness. All of these categories had clear criteria and decision processes.

508 buffer-related warnings and 153 input validation warnings were reported for the C/C++ systems (in total there were seven security categories). For what the SATE team call "well known and well studied categories" such as buffer-related security flaws the overlap of tool reports was higher. The security reports for the Dovecot system had a 50 % overlap in total. As for the CVE-selected test cases the tools had problems finding them and a summary of Chrome's nine CVEs provides some explanations such as an assertion that aborts in debug mode confusing the tools.

No explicit results per tool were published in the paper since the purpose was not to find "the best" tool.

Further Surveys of Static Analysis Tools

Tevis and Hamilton presented a theoretical survey of 13 static analysis tools aimed at security—BOON, CodeWizard, FlawFinder, Illuma, ITS4, LDRA, MOPS, PC-Lint, PSCAN, RATS, Splint, UNO, and WebInspect [32]. Several of these tools were covered in our empirical survey, see Paper C. Tevis and Hamilton argue that the deeper issue of insecure code lies in imperative programming and that a paradigm shift towards functional programming techniques could hold the key to removing software vulnerabilities altogether.

Zitser et al published an empirical survey of static analysis tools run on vulnerable and patched versions of open source systems BIND, WU-FTPD, and Sendmail [33]. The vulnerable versions contained 14 known exploitable buffer overflows. The analysis tools evaluated were ARCHER, BOON, PolySpace, Splint, and UNO. True and false positives were found to be:

- PolySpace: 87 % true positives, 50 % false positives
- Splint: 57 % true positives, 43 % false positives
- BOON: 5 % true positives, 5 % false positives
- ARCHER: 1 % true positives, 0 % false positives
- Uno: No warnings concerning buffer overflows

Chess and McGraw wrote a short theoretical review of static analysis for security, covering the tools BOON, CQual, xg++, Eau Claire, MOPS, and Spint [34]. Their main focus is on important properties of such tools such as ease of use and completeness of rule sets.

Kratkiewicz did her Masters Thesis work on evaluating static analysis tools against a buffer overflow testbed [35] and the work was also published in a paper **together with Lippmann** [36]. 291 small C programs called *test cases* were used to evaluate five static analysis tools—ARCHER, BOON, PolySpace, Splint, and Uno. Interestingly that's the same lineup as Zitser et al used a year before. Kratkiewicz and Lippmann's results were:

- PolySpace: 99.7 % true positives, 2.4 % false positives
- Splint: 56.4 % true positives, 12 % false positives
- BOON: 0.7 % true positives, 0 % false positives
- ARCHER: 90.7 % true positives, 0 % false positives
- Uno: 51.9 % true positives, 0 % false positives

Interestingly only true positives for PolySpace and Splint match well or fairly well between the Zitser et al and Kratkiewicz and Lippmann studies. All the other results differ heavily. Kratkiewicz and Lippmann comments on this—"Good performance on test cases (at least on the test cases within the tool design goals) is a necessary but not sufficient condition for good performance on actual code." It should be noted that Lippmann is one of the co-authors of the Zitser et al paper.

Zheng et al analyzed the effectiveness of static analysis tools by looking at vendor tests and customer-reported failures for three large-scale network service software systems at Nortel Networks [37]. Three tools were used—FlexeLint, Reasoning's Illuma, and Klocwork's inForce. The tools were not compared. On the contrary the authors based most of their analysis on the output of FlexeLint since it had the highest number of reported faults and the greatest fault variety. Zheng et al concluded that static analysis tools are effective at identifying code-level defects such as assignment and checking faults and an affordable means of software fault detection. However, other techniques such as manual inspection is needed to detect more complex, functional, and algorithmic faults.

Michaud and Carbone published a technical report called "Practical verification & safeguard tools for C/C++" [38]. Their study did not only cover static analysis tools but in that category they did empirical evaluations of the tools PolySpace, Coverity Prevent, Grammatech CodeSonar, and Klocwork K7. They augmented an existing open source program with synthesized defects—"synthetic tests"—and they used an existing numerical analysis application known to be buggy and badly designed—"production code tests". In the category most closely related to our study—"Overrun and Underrun Faults" in synthetic tests—the results were:

- PolySpace: 55.6 % true positives, 37.5 % false positives
- Coverity: 55.6 % true positives, 0 % false positives
- CodeSonar: 55.6 % true positives, 0 % false positives
- Klocwork K7: 77.8 % true positives, 0 % false positives

As for the production code tests Michaud and Carbone could not get good results for any of the tools under evaluation. They suspect low-quality code such as the numerical analysis application they used is too hard to analyze for the tools, but could not prove that was the case. The poor results made the authors uncertain of their test setup and thus they never published the exact results of the production code tests.

Baca et al evaluated the static analysis tool Coverity Prevent for cost reduction in industrial software engineering [39]. Three C++ software products, proprietary telecom and open source were used as testbed.

The paper makes no distinction between security or non-security issues in false positives which means the outcome cannot really be compared with similar studies. The security-related results were:

- Product A, 600 KLOCs: 37.5 % true positives out of 8 known issues, 82 new true positives, and 22.1 % false positives (both security and non-security)
- Product B, 500 KLOCs: 28.6 % true positives out of 7 known issues, 5 new true positives, and 5.3 % false positives (both security and non-security)
- Product C, 50 KLOCs: 25 % true positives out of 8 known issues, 7 new true positives, and 6 % false positives (both security and non-security)

The authors' primary goal was to measure potential cost reduction and the results showed that on average 17 % could be saved if static analysis tools were used.

Kupsch and Miller published an evaluation of manual versus automated security assessments [40]. The system under study was Condor, a workload management system for compute-intensive jobs. Condor is written in C. The static analysis tools used were Coverity Prevent and Fortify SCA. 15 serious security flaws were found by manual inspection. 6 of these were found by Fortify and only one by Coverity. The two tools did report thousands of potential defects but the authors could not find any severe security flaws among them except the ones already found in manual inspection.

Johns and Jodeit have developed a methodology for evaluating or surveying security-targeted static analysis tools [11]. Their choice of a micro benchmark approach was based on Chess and West's four criteria—Quality of the analysis, Implemented trade-offs between precision and scalability, Set of known vulnerability types, and Usability of the tool [41].

In their implemented setup they have every testcase in a separate, dedicated application containing either a true vulnerability or a crafted false positive. These testcases are made into executable units by being forged with a host program. All testcases for a given programming language share the same host program. False positives due to the host program's code are eliminated by an analysis of the host program itself plus a diff. The host program contains all the infrastructure required by the testcases, for instance a simple TCP server that reads untrusted network data and passes it to the testcases.

Testcases fall into one of three categories—Vulnerability class coverage (e.g. does the tool check for buffer overflows?), Language feature coverage (e.g. does the tool still distinguish between safe and unsafe buffer access when combined with advanced scoping rules?), and Control- and data-flows (e.g. will loop invariants be considered when checking the data flow from a source to a sink?).

Non-disclosure agreements prohibited Johns and Jodeit to publish empirical results from commercial static analysis tools. However, they came to some general results, two of which are relevant our scope:

- Tools tend to favor soundness over low false positive rates
- Tools checking C code did well warning for double `free()` and `null` dereferences but had significant problems with non-trivial integer overflow vulnerabilities

An overview of true and false positives for all the empirical studies above together with our's from 2002 is presented in Figure 3.1.

Real-World Versus Synthesized Comparisons

In 2008 **Emanuelsson and Nilsson** published a comparative study of industrial static analysis tools [42]. They compare the effectiveness and efficiency of the tools PolySpace, Coverity, and Klocwork on industrial software at the digital communications company Ericsson.

While not focused only on security a number of their results are interesting given the approach we took with a synthesized testbed in our 2002 comparative study (Paper C), namely:

- The rate of false negatives, i.e. actual bugs missed, is very difficult to estimate given that the total number of bugs is unknown.

		Wilander, Kamkar	Zitser et al	Kratkiewicz, Lippmann	Michaud, Carbone	Baca et al
Flawfinder	True pos.	96 %				
	False pos.	71 %				
ITS4	True pos.	91 %				
	False pos.	52 %				
RATS	True pos.	83 %				
	False pos.	67 %				
Splint	True pos.	30 %	57 %	56.4 %		
	False pos.	19 %	43 %	12 %		
BOON	True pos.	27 %	5 %	0.7 %		
	False pos.	31 %	5 %	0 %		
PolySpace	True pos.		87 %	99.7 %	55.6 %	
	False pos.		50 %	2.4 %	37.5 %	
ARCHER	True pos.		1 %	90.7 %		
	False pos.		0 %	0 %		
Uno	True pos.			51.9 %		
	False pos.			0 %		
Coverity	True pos.				55.6 %	30.4 %*
	False pos.				0 %	
CodeSonar	True pos.				55.6 %	
	False pos.				0 %	
Klocwork	True pos.				77.8 %	
	False pos.				0 %	

Table 3.1: Overview of five comparative studies on static analysis tools for security. Upper percentage in each cell gives the true positive rate. Below is the false positive rate. *Average from analysis of three systems.

- Different tools analyzing the same codebase typically had a low overlap in reported bugs, both true and false positives. For one piece of software Klocwork reported 32 defects including 10 false positives, Coverity reported 16 defects including one false positive, and only three defects overlapped between the reports.

- In two cases of analyzing software with known bugs none of the tools found any of them.

These results show the importance of not only evaluating tools on real-world code but also on synthesized testbeds, i.e. controlled environments. We've taken the approach of implementing controlled testbeds in three of our studies, see Papers C, F, and G.

Buffer Overflow and Format String Attack Prevention

Our comparative study from 2002 covered static analysis tools trying to prevent buffer overflows and format string attacks (Paper C). Additionally, our proposed new formalism for modeling and pattern matching security properties of code was built up on dependency graphs and GrammaTech's tool CodeSurfer (Papers D and E). This makes our research very closely related to **Nagy and Mancoridis'** research on static analysis with dependency graphs and CodeSurfer to find buffer overflow and format string flaws [43]. Nagy and Mancoridis also introduce interesting metrics on how to prioritize reported flaws, an important issue that we addressed too in our paper on modeling security properties of code, Paper D.

Their analysis of code takes the following approach:

1. Define all I/O system calls as sources of potentially malicious input (henceforth *user input*). Formally 28 functions from the *C standard library* and parameters to the program's `main` function.

2. Perform dataflow analysis to determine where user input can reach. The union of all reachable paths defines the code to analyze.

3. Calculate two metrics for ranking output in developer feedback—*coverage* and *distance*. Coverage is defined as the percentage of a function's statements that handle user input. Distance is defined as the shortest path of dependency graph nodes between the source of user input and the start of the given function. Such dataflow paths are built up of re-assignments and modifications of user input.

4. Use a fault detection mechanism similar to Livshits and Lam's IPSSA technique (see Paper D) and pattern matching of known sinks to detect potential buffer overflows and format string flaws. `strcpy` and `sprintf` are examples of buffer overflow sinks, and the `printf` family of functions are the natural format string sinks.

5. Use control dependencies to detect if there's a conditional statement checking the size of the user input before copying it to a buffer. Such a conditional will be deemed as avoiding the buffer overflow. This is the same basic approach we use in Paper E but without the ranking metric of the various ways such input validation can go wrong.

They analyzed 12 security critical open source products comprising over 800 KLOCs. Out of these they gave extra focus on the largest piece of software, the Pidgin chat client with its 230 KLOCs. The total input coverage of the Pidgin code was 10.56% and the longest shortest path from source to sink was 100 graph nodes. One flaw was detected, a buffer overflow where the name of the user's home directory could overflow a 128 character sized buffer.

Graph Reachability

In 2008 **Scholz et al** introduced a static analysis technique for computing user input dependencies [44]. They use both data and control flow dependencies in an augmented Static Single Assignment form.

Inter-procedural dependencies are handled in one of two ways—call-insensitive (less precise but fast) and call-sensitive (more precise but slower). In the call-insensitive case the whole program under analysis is modeled in a single reachability graph. In the call-sensitive case the program's call graph is split into strongly connected components containing functions that potentially call each other recursively, reachability is computed for each such component, and summary functions are marked as user input dependent if they use tainted arguments, globals, or results of tainted functions.

They have implemented their approach using a low-level virtual machine for C. A configuration file specifies which globals, arguments, and results of functions that are user input dependent.

In their empirical evaluation they analyze nine C programs—sendmail, httpd, perlbmk, vortex, pppd, sshd, mailx, zoneadmd, and mail. In the case of call-sensitive analysis the percentage of user input dependent code varied between 77.7 % for httpd and 42.1 % for pppd. This is interesting comparing to Nagy and Mancoridis' results presented in Section 3.1.1.

The results as to finding real security bugs are not presented in the paper.

3.1.2 Model Checking

Model checking is a technique for automatically verifying correctness properties of finite-state systems (brief introduction on Wikipedia [45]). The technique was pioneered by **Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis** and earned them the 2007 Turing Award [46].

Software model checking is used for verifying software correctness and needs both a formal model of the software and of its specification. Such formal models can be built using well-known abstractions, for instance abstract syntax trees [47]. Software model checking may fail to prove or disprove a given system property due to undecidability problems such as the famous halting problem.

Model Checking Versus Static Analysis

Engler and Musuvathi presented their experiences on static analysis versus software model checking in an invited paper at VMCAI 2004 [48]. According to their studies model checking did not find more bugs in software than static analysis. Specifically, checking FLASH cache coherence protocol code was four times as effective with static analysis in terms of found bugs. They conclude that a major challenge of software model checking is to correctly and completely model the environment in which the software to be checked operates.

Model Checking Securty Protocols

Model checking has been used extensively to verify security protocols such as key exchange protocols. The attacks in scope for such model checking include *man-in-the-middle* (attacker imposing between two communicating parties), *reflection* (exploiting protocol symmetry by bouncing messages back), *oracle* (trick an honest agent into revealing information), *replay* (attacker reusing earlier messages), and *interleave* (exploiting overlaps between two or more protocol runs) [49].

Early work on using model checking for security protocols was done by **Mitchell et al** in 1997 when they analyzed the Needham-Schroeder protocol, variants of Kerberos, and the faulty TMN protocol [50].

Model Checking Code Security

Chen and Wagner have designed a model checking tool called MOPS which checks for security bugs that can be described in terms of temporal safety properties, i.e. ordering constraints [51]. MOPS is briefly presented in Paper D.

In 2005 **Schwarz et al** presented the results of model checking the entire Red Hat Linux 9 distribution for security violations using MOPS [47]. The scope was 60 million lines of code and they discovered 108 exploitable bugs.

Ku et al developed and published a buffer overflow benchmark for software model checking [52]. It consists of 298 code fragments produced from 22 vulnerabilities and corresponding patches in 12 open source programs. They could only find one publicly available model checker that soundly models C arrays—SatAbs. Buffer sizes were limited to 1 and 2 for the sake of reasonable execution time. At buffer size 1 SatAbs found an overflow in 71.4 % of the testcases.

3.2 Security Requirements Engineering

There's a vast amount of research done in the area of security requirements engineering since our latest publication in 2007. New methods, formalisms, categorizations, and cross-cutting studies continue to get published. However, this section focuses on empirical studies of requirements engineering practice for security since that is the topic of our Paper A and B.

3.2.1 Eliciting, Analyzing, and Documenting Security Requirements in Practice

Elahi et al have carried out an interview study to collect how businesses elicit, analyze, and document security requirements in practice. 400 software professionals, mostly programmers, from 237 firms in China (both Chinese and international) where asked 18 multiple choice questions via a web-based survey. 374 of them took part.

The goals of the study were to discover in which development stages security was considered, the approach to elicitation, modeling, and documentation, knowledge sources exercised, and how risk assessment was done.

Several of their findings were interesting:

- 55 % had received security training during education and 27 % in their professional life. Only 17 % had no security training at all.
- 48 % use security standards or guidelines such as ISO 17799, Common Criteria, and SANS guides.
- Only 9 % said they explicitly documented security requirements while 59 % said they consider security requirements implicitly. 31 % did not elicit security requirements at all.
- 88 % stated that security may lead to trade-offs, most often conflicting with performance, efficiency, or specific features.
- 82 % said they analyze security risks and when given the choice of various metrics or scales over 80 % picked a scale with low granularity (low/medium/high or 1-9).
- A statistical analysis showed no correlation between the respondents roles and their approach to security requirements elicitation.

3.2.2 A Survey of Security Requirements Methodologies

Tøndel et al published a literature survey on security requirements methodology covering both academic publications and popular books written by industry practitioners [53]. They conclude that no common agreement exists on what a security requirement is. According to Tøndel et al the approaches under study ...

- don't agree on if and how requirements should state concrete security measures;
- provide different levels of detail as to how to perform the tasks; and
- require different levels of expert knowledge.

Product owners and project leaders are often unaware of what expertise is needed to properly elicit and specify security requirements as shown in our empirical research (see Papers A and B). Tøndel et al show that even a chosen method will not solve that problem. Further research has to be done on what security requirements are and how they should be elicited and specified.

Chapter 4

Reflections

Much has happened during the ten years that have passed since our first empirical experiments on buffer overflow prevention. Software is by no means becoming less important for mankind. On the contrary, more and more of our societies and lives are dependent on working software. Additionally, the global inter-connection between computers has become yet another foundation for much of what we use software for.

New types of software vulnerabilities have been discovered during these years, for instance *clickjacking* [54]. Others have grown from discussions in the security community to worldwide problems, for instance *cross-site scripting*, first published in a CERT advisory February 2000 [55] and now in second place on OWASP Top 10 of security risks in web applications [56].

4.1 Static Analysis

Since 2002, static analysis tools for security has become a full-blown industry with several tool vendors and business acquisitions by enterprises like IBM and HP. Many software developing organizations now use static analysis for the purpose of software security. Looking back at our research efforts in 2002-2005 we see that industry has favored few false positives over few false negatives and that software vulnerabilities are still mostly reported as text rather than in some visualized form.

Our proposal to use dependency graphs is still valid in terms of modeling good and bad programming practice. However, it remains an open question if it is an efficient formalism for scanning code. Especially considering the difficulty of building system-wide dependency graphs for applications with a client written in one language (e.g. JavaScript) and a server system written in another or several other languages (e.g. Java, SQL, and Cobol).

4.2 Security Requirements

In our experience, security requirements practice is still based on top lists of vulnerabilities and the knowledge among developers rather than among product owners. The significant change that we have seen is the rise of software security compliance requirements, notably in the *Payment Card Industry Data Security Standard* (PCI DSS) [57], and in security breach notification laws. Most of U.S. states have breach notification laws [58] and the European Union has one in the Directive on Privacy and Electronic Communications [59].

Another development in security requirements is so called *threat modeling*, popularized by the Microsoft book Security Development Lifecycle [8] and wider in scope than Bruce Schneier's *Attack Trees* [60]. Threat modeling is a structured, iterative process for assessing security threats to applications and systems based on their security objectives (see Figure 4.1). Identified threats are mapped to suspected vulnerabilities. In its pure form threat modeling does not cover countermeasures and thus not detailed security requirements. However, both the identification of security objectives and the identified threats and vulnerabilities tie in with the work on security requirements.

Looking back at our empirical study on current practice in security requirements it is clear that industry has not decided to send all IT project leaders and product owners to security training. Nor have they developed a standard set of security requirements to pick from. Instead security (outside compliance) is mostly handled by security experts and developers who take pride in writing robust, secure software.

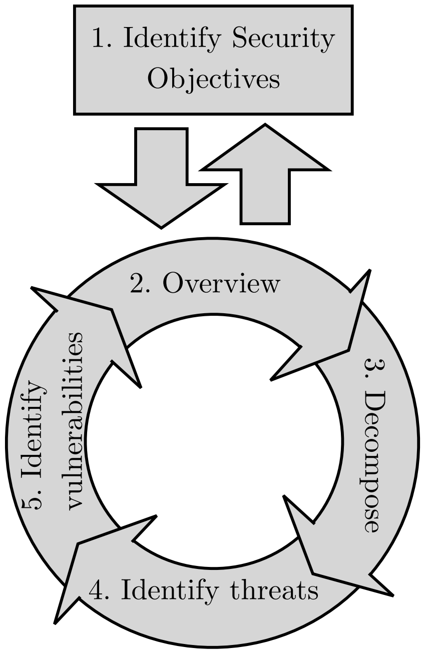


Figure 4.1: **Iterative Threat Modeling**. Image recreated from the Microsoft Developer Network article on threat modeling web applications [61].

Additionally, we've experienced a tendency towards business developers not wanting secure software *per se*. Rather, they want attackers and criminals to go away, either to jail or just quit doing their evil deeds. If they can not get rid of the attackers, only then does software security and its investments make sense. There is an important lesson to be learned from this. In all parts of society there has to be a balance between crime rates and reasonable crime protection and awareness. If cybercrime keeps increasing indefinitely, the Internet and computer-based business will be left to vigilantes and security pros which is far from where we want to be. Where to draw the line and get the right balance is an open question.

4.3 Runtime Intrusion Prevention

Systems protecting themselves, or runtime intrusion prevention, continues to be a very active area of research. There is an ongoing race between such protection and intrusive techniques. For instance a so called stack smashing buffer overflow attack has become almost impossible if the target system has current countermeasures such as non-executable memory, a randomized 64-bit address space, stack integrity checks, application sandboxing etc. But almost every assumption of the stack smash attack has been challenged since Aleph One's seminal paper "Smashing the Stack for Fun and Profit" back in 1996 [62].

Perhaps the most notable disruption was caused by Hovav Shacham's introduction of *return-oriented programming* (ROP) in 2007 [63]. Suddenly an attacker did not need to a) inject attack code, b) execute code in non-executable memory segments, and c) enforce a malicious call to a libc function. In the years since, several publications have suggested countermeasures and in 2012 Vasilis Pappas from Columbia University published his ROP protection called kBouncer [64] with which he won Microsoft's BlueHat Prize 2012 of USD 200,000 [65]. The BlueHat Prize is awarded to the most "novel runtime mitigation technology solution that is capable of preventing the exploitation of memory safety vulnerabilities".

This shows how the discussion on first *full disclosure* and then *responsible disclosure* has come full circle and the security community is now invited to compete for the best intrusion prevention technologies, and not only collecting bug bounties.

Our research in runtime intrusion prevention has been focused on language-level deficiencies (e.g. no buffer bounds checking in standard C). In parallel there has been research on runtime intrusion prevention where application and even business logic is taken into consideration, for instance in the OWASP AppSensor project [66]. One of their ideas is to apply basic integrity checks of incoming parameters to a web application by validating the number of parameters, the allowed characters in input fields, and the allowed values of enumerations (e.g. drop-down lists). Such checks makes it much harder to exploit vulnerabilities deeper in the application just like non-executable memory and canary values have made buffer overflow attacks harder.

4.4 Thoughts on the Future of Intrusion Prevention

Intrusion prevention has come far, especially in protecting memory safety in operating systems. But attacks against applications are not decreasing. It is our belief that lack of data modeling and lack of support for data modeling lies at the heart of the software security problem.

So much of the data transferred between systems and humans is modeled by basic datatypes, primarily strings. Very seldom do these pieces of data mean just “any string”. Instead they are part of a business or organizational data domain with quite specific meanings such as last name, medication, or search criteria. Our failure to model what is and what is not a human name or the name of a medication opens up for malicious injection of commands, code, syntactic errors, semantic errors, and even random data. Proper data modeling is hard, but to make an application withstand the input of “any string” while maintaining a secure state is arguably harder.

The lack of data modeling manifests itself by the prevalence of parsers, encoding, and decoding. As soon as data arrives at a platform (e.g. a web browser), a client, or a server it has to be interpreted. This often means parsing strings, decoding strings, and encoding strings. So while the data might have had a specific, well-defined meaning at its origin, it is flattened out into a general string during transit and the meaning has to be resurrected upon arrival. Protecting string parsing, encoding, and decoding from malicious manipulation is hard but all too often left to the application programmer. If we instead could support programmers with tools for end-to-end data modeling we would stand a much better chance against attackers.

Paper A

Security Requirements—A Field Study of Current Practice. Published in the Proceedings of the Symposium on Requirements Engineering for Information Security 2005 (SREIS). The layout is modified.

Security Requirements—A Field Study of Current Practice[1]

John Wilander and Jens Gustavsson
Dept. of Computer and Information Science, Linköpings universitet
{johwi, jengu}@ida.liu.se

Abstract

The number of security flaws in software is a costly problem. In 2004 more than ten new security vulnerabilities were found in commercial and open source software every day. More accurate and consistent security requirements could be a driving force towards more secure software. In a field study of eleven software projects including e-business, health care and military applications we have documented current practice in security requirements. The overall conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions. We show how the requirements could have been enhanced by using the ISO/IEC standard for security management.

Keywords: security requirements, non-functional requirements

1 Introduction

According to statistics from CERT Coordination Center, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software [67]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [68].

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education, Vinnova 2GAP (Second Generation Application Provisioning), and EU EASYCOMP (Easy Composition in Future Generation Component Systems).

For consumers of software the security of the products they use relies heavily on the security requirements specified for the products. If these requirements are poorly specified there is nothing saying that the producers will strive for security. Instead, costs and time will be focused on meeting the other requirements, and security issues may be left for maintenance in the infamous *penetrate and patch* manner [69].

To build more secure software, accurate and consistent security requirements must be specified. We have investigated current practice by doing a field study of eleven requirement specifications on IT systems being built 2003 through 2005. To evaluate the outcome we have looked into documentation of security requirements from the requirements engineering community as well as from the security community. Requirements found in the specifications have been categorized into security areas and divided into functional, non-functional, and assurance requirements. The ISO/IEC standard for security management has been used as an example of how a standard could help to specify better security requirements.

The rest of this paper is organized as follows. In Section 2 we look at how security requirements have been defined within the requirements engineering community and the security community. Next, Section 3 discusses security testing to verify that security requirements have been met. Section 4 presents and discusses our field study of eleven requirements specifications and what they specify in terms of security. Finally, Section 5 concludes our work.

2 Security Requirements

A subgroup of software requirements is security requirements. A lot of work and research has been done to define and standardize security requirements, especially by military organizations. Here we look at (examples of) how security requirements are defined within the requirements engineering (RE) community and the security community.

2.1 From a RE Point of View

Within requirements engineering security is often conceived as a non-functional requirement along with such aspects as performance and reliability, and is generally considered hard to manage [70, 71, 72, 73].

There are several (partially overlapping) definitions of functional and non-functional requirements. The one used in this paper is based on the IEEE definition [74], Thayer and Thayer's glossary [75], extended by Burge and Brown [70].

Functional Requirement. A functional requirement (*FR*) defines something the system must do, capturing the nature of the interaction between the component and its environment. A FR must be *testable*, which means it is possible to demonstrate that the requirement has been met by a test case resulting in pass or fail [70, 74].

Non-Functional Requirement. A non-functional requirement (*NFR*) is a software requirement that describes not what the software will do, but how the software will do it. NFRs restrict the manner in which the system should accomplish its function. NFRs tend to be general and concern the whole system, not just some parts [70, 75].

In their paper on the future of software engineering Premkumar Devanbu and Stuart Stubblebine discuss security requirements. They define them as:

Security Requirement. A security requirement is a manifestation of a high-level organizational policy into the detailed requirements of a specific system [73].

2.2 From a Security Point of View

One of the seminal documents on security requirements is the *Common Criteria*, or *CC*. The CC is a standard and is meant to be used as the basis for *evaluation* of security properties of IT systems [76].

> "The CC will permit comparability between the results of independent security evaluations. It does so by providing a common set of requirements for the security functions of IT products and

> systems and for assurance measures applied to them during a security evaluation."

Following the CC standard, consumers of software produce a *Protection Profile* that identifies desired security properties of a product. The Protection Profile is a list of security requirements. Producers on the other hand create a *Security Target* that identifies the security-relevant properties of the software. A Security Target can meet one or more Protection Profiles. CC distinguishes between two types of security requirements—functional and assurance:

Security Functional Requirement (CC). Security functional components express security requirements intended to counter threats in the assumed operating environment. These requirements describe security properties that users can detect by direct interaction with the system (i.e. inputs, outputs) or by the system's response to stimulus.

Security Assurance Requirement (CC). Requiring assurance means requiring active investigation which is a process requirement. Active investigation is an evaluation of the IT system in order to determine its security properties.
Common Criteria lists what can be done in terms of assurance through evaluation. We highlight a few things here to give an example of what these requirements can look like:

- Analysis and checking of process(es) and procedure(s);
- checking that process(es) and procedure(s) are being applied;
- analysis of functional tests developed and the results provided;
- independent functional testing; and
- penetration testing.

Another relevant standard is the *ISO/IEC 17799 Information technology—Code of practice for information security management* [77]. The section on "Systems development and maintenance" includes ten pages specifying requirements and explaining considerations for techniques such as input validation, encryption, and security of system files.

The ISO/IEC standard does not discuss functional, non-functional, or assurance requirements as such.

3 Security Testing

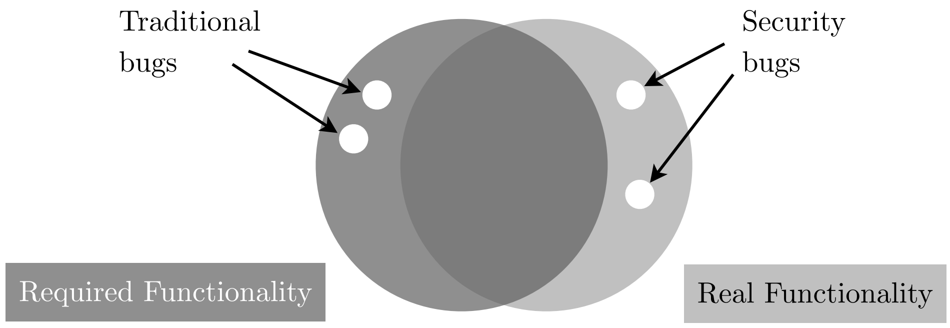


Figure 1: Finding security bugs through testing often means testing for side-effects and functionality outside the requirement specification.

Closely related to requirements is testing. If something is considered a requirement there needs to be some way to verify that it has been met. This can be done with testing where the outcome is pass or fail.

"Traditional" bugs are deviations from the requirement specification, either by doing B when supposed to do A, or by only doing B when supposed to do A and B.

Thompson and Whittaker write about running test cases to find security bugs [78]. Such bugs often differ from traditional bugs by being hidden in side effects. Finding security bugs means finding out what the system *also* does, apart from the specified functionality. Thompson and Whittaker's Venn diagram shows this (see Figure 1).

Requirements on absence of side effects are typically non-functional. Specifying what the system must not do clearly restricts in what way the functional requirements can be fulfilled. Moreover, requirements on *testing* of side effects are not only non-functional but also a kind of security assurance requirement.

This stresses that we need non-functional requirements, and specifically security assurance requirements to specify more secure systems. As we will see later such requirements are rare in current practice (see Section 4).

4 Field Study of Eleven Requirements Specifications

We have studied eleven requirements specifications of IT systems being built 2003 through 2005. In this section we first present an overview of security areas found in the specifications, and an overview of the systems and organizations that have written the specifications. Next, we present both a summarized and a detailed categorization of all security requirements found. The categorization is done into security areas and into functional, non-functional, and assurance requirements. Finally, we discuss the outcome and reflect on potential shortcomings in the material.

On an abstract level we have categorized the security requirements into well-known security areas. A full description along with examples for each category can be found in Internet security glossaries [79, 80].

4.1 Systems in the Field Study

In our study we have taken advantage of the fact that all requirement specifications used for public procurement by Swedish Government or local authorities are public documents. The authorities are also required by law to publish their requests for tenders, and all such requests are categorized depending on the type of products or services bought. The categorization is called Common Procurement Vocabulary (CPV), which is a European standard [81].

We used a commercial database to find "Computer and related services" purchases made by Swedish Government or local authorities from January 2003 to June 2004 [82]. In Table 1 you find a summary of all security requirements found. Here is a brief description of the systems studied:

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Access Control/Roles	1	11	6	5	8	5	4	5	3		3
Attack Detection							2	4		3	
Backup		5	9	2			2	2			
Digital Signatures			1		1	1	1	2			1
Encryption							4	1			1
Integration							2	1			
Logging		9	3	1	11	1	5	8	1		
Login		5	3	3	8	2		2	1		2
Privacy			2							1	
Authentication							2	4	2		1
Availability	1		3			1	6	4		3	1
Design/Implementation					1			6			1
Physical Security								6			
Risk Analysis										1	
Security Management								2		2	
Security Testing								1			

Table 1: Overview of security requirements on eleven IT systems being built during 2003-2005. The double horizontal line divides the requirement categories into mostly functional (above) and mostly non-functional (below). Figures tell how many requirements were found in each category.

Billing (City of Jönköping). A billing system for drinking water, sewage, and garbage collection.

Accounting (Cities of Dalsland). System for handling ledgers, accounting, and budgets for five cities in the province of Dalsland.

Salary/Staff 1 (The cities of Kinda, Ödeshög, Boxholm, and Ydre). System for administration of salaries and staff within the cities.

Salary/Staff 2 (The cities of Stenungsund and Tjörn) System for administration of salaries and staff within the cities.

E-Business (The cities of Skövde, Falköping, Karlsborg, Mariestad, Tibro, Tidaholm, and Hjo). System for electronic trade and business including billing.

Defense Materiel (Swedish Defence Materiel Administration). Web-based marketplace for consulting services to the Swedish Armed Forces.

Medical Advice (The Federation of County Councils). System for managing medical advice by phone on a national level. Redirection of calls, queue management, work-flow management, medical documentation, and statistics.

Health Care 1 (Stockholm County Council). Integration platform to support personal medical information following patients between various health care organizations.

Health Care 2 (The city of Lomma). System for event handling in health care including personal medical records.

Highway Tolls (The City of Stockholm's Executive Office). Equipment, systems and services for handling environmental fees for all vehicles entering the city of Stockholm.

Hazmat (Swedish Maritime Administration). Ship reporting system managing mandatory reporting of hazardous goods, arrival, departure, and generated waste in accordance with EU directives.

4.2 Detailed Categorization of Security Requirements

In tables 2, 3, 4, 5, and 6 we present the complete list of security requirements found in the specifications. The list is divided into security areas and every requirement is categorized as functional, non-functional, or security assurance (subcategory of non-functional). The numbers in the table are the number of requirements found for each subcategory. For instance the "E-Business" system has four specific requirements on access control per person (see Table 2).

The security areas are conventional but the categorization relies on the fact that the authors of the specifications know how the various terms differ,

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Access Control/Roles											
- per person (FR)	1	4	3	2	4	3	2		1		1
- per group (FR)		1	1	1	2	2	1	1			1
- one person many roles (FR)				1	1		1				
- file access r/w/x (FR)		6	2	1				4	2		1
- role-based GUI (FR)					1						
Attack Detection											
- intrusion detection (FR)							1	2		1	
- fraud detection (FR)										2	
- antivirus (FR)							1	2			
Backup											
- in general (FR)		1	4				1				
- automatic (FR)		3	2	1			1				
- time interval (FR)		1		1							
- durability (NFR)			2								
- data versioning (FR)								2			
- done run-time (FR)			1								

Table 2: Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

for instance the difference between access control, authorization and login where we have found similar requirements in all categories.

It is important to note that these are the requirements found in the specifications, thus *not* a complete list of possible security requirements. For a complete list we refer to published standards such as Common Criteria [76] and ISO/IEC standard for security management [77].

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Digital Signatures											
- in general (FR)					1	1		1			
- use of standard (NFR)								1			1
- use of PKI (FR)			1								
- for data origin (FR)							1				
Encryption											
- use of standard (NFR)							1				1
- during login (FR)							1				
- filesystem (FR)							1	1			
- network traffic (FR)							1				
Integration											
- with firewall (FR)							1				
- with anti-virus (FR)							1				
- with external PKI (FR)								1			
Logging											
- in general (FR)		6	1	1	1	1		1			
- automatic (FR)							3	3			
- what info to be logged (FR)		3	2		8			2			
- log not changeable (FR)					1		2	2	1		
- tool for log analysis (FR)					1						

Table 3: (Continued) Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

4.3 Discussion

Data from the field study show that—(1) Security requirements are poorly specified, and (2) The security requirements specified are mostly functional.

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Login											
- username, password (FR)		2		1		1					1
- password change (FR)		2	1	1	2						1
- smart card (FR)								1			
- Single Sign-On (FR)			1	1	1			1	1		
- automatic logout (FR)		1	1		1	1					
- non-guessable passwords (FR)					1						
- resticted login attempts (FR)					1						
- inactivate old accounts (FR)					1						
- password re-use (FR)					1						
Privacy											
- anonymity (FR)										1	
- classification (FR)											

Table 4: (Continued) Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

Security Requirements are Poorly Specified

To support the conclusion that the security requirements are poorly specified we highlight three things:

1. Inconsistent selection of security requirements
2. Inconsistent level of detail
3. Security standards are not required

Inconsistent Selection of Security Requirements. In several of the specifications studied we note that some relevant security areas are fairly well specified whereas other are completely left out. Typically, a need for

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Authentication											
- use of standard (NFR)								3			1
- per person (NFR)							1				
- per system/entity (NFR)							1	1			
- smart card (FR)									1		
- biometrics (FR)									1		
Availability											
- 24h/day, 7 days/week (NFR)	1					1		1			1
- precentage uptime (NFR)			1					1		2	
- redundant power and net (NFR)			2				3	1			
- redundant data (NFR)							3			1	
- automatic restart (FR)								1			
Design/Implementation											
- compartmentalize (NFR)								1			
- input validation (NFR)								1			
- output validation (NFR)								1			
- referential integrity (NFR)					1			1			
- file integrity (NFR)								2			
- fault tolerant interfaces (NFR)											1

Table 5: Detailed categorization of mostly non-functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional, and (SAR) means security assurance (subcategory of non-functional).

security has been expressed with detailed functional security requirements whereas non-functional requirements are left out. This may lead to security problems (see Section 3).

Examples of such inconsistencies can be seen in access control/roles where all systems have requirements (two referring to standard) which indicates that restricted access is important. At the same time only three

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Physical Security											
- in general (NFR)								1			
- fire (NFR)								2			
- water/moist (NFR)								1			
- physical intrusion (NFR)								2			
Risk Analysis											
- fraud risk (SAR)										1	
Security Management											
- use of ISO/IEC standard (SAR)								2		2	
Security Testing											
- availability, stress test (SAR)								1			

Table 6: (Continued) Detailed categorization of mostly non-functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional, and (SAR) means security assurance (subcategory of non-functional).

specifications require some kind of encryption of data communication and only two specifications require physical security including restricted physical access.

Inconsistent Level of Detail. Some security requirements have a high level of detail whereas others in the same specification are only specified on a general level. This might indicate that the organizations specifying the security requirements rely heavily on local competence and not standards.

We call this phenomenon *local heroes*—for instance, there might be someone who knows very much about backup systems and thus the specifications on backup become detailed and fairly complete. But in other security areas the organization does not have an expert, which leads to under-specified requirements in that area.

This phenomenon can be seen in for instance the "E-Business" system where the requirements on logging are very detailed (eight requirements on what info to be logged) and at the same time digital signatures are specified as "The system should be able to handle the use of electronic signatures" with no further details.

In the specification of "Salary/Staff 1" we find detailed requirements on backup (automation, durability, and run-time backup), while in the same specification the lone requirement on digital signatures is "The system should handle electronic signatures and interfaces to PKI cards etc".

Security Standards are Not Required. Many security areas have well-known and rigorously reviewed standards such as encryption and access control policies. The specifications studied very seldom require these standards to be followed. Instead the requirements specified leaves to designers and implementers to choose or even invent the technology to be used. Such an ad-hoc approach to security is known to lead to problems [69].

None of the specifications explicitly requires a standard policy for access control. In the case of digital signatures two out of six specifications explicitly require a standard solution. And for the area attack detection no publicly known system is required which means the producer can implement his/her own anti-virus software etc.

Security Requirements are Mostly Functional

As mentioned in Section 2.1, security is often conceived as a non-functional requirement, and as such it is known to be hard to manage. However, our study shows that in more than 75% (164 out of 216) of the cases, security requirements boil down to functional requirements. This transformation of abstract non-functional requirements into concrete functional requirements is known and resembles Chung *et al's* technique of "refining initial high-level goals to detailed concrete goals" [71].

However, the kind of non-functional security assurance requirements discussed in Section 3 are left out in almost all cases—we identified 6 such requirements out of 216. The security areas risk analysis, standardized security management, and security testing were categorized as security assurance. The overall distribution of requirements is; CC's security functional requirements divided into functional (76%) and non-functional (21%), and last CC's security assurance requirements as non-functional (3%).

Security Requirements Absent

A natural question is—what security requirements are left out in the specifications studied? Since we decided to list only the requirements present in at least one specification, a comparison with a more complete list would indicate what could be gained. A fair comparison can be made in terms of level of detail. If a security requirement is specified it is unlikely that it has been deliberately under-specified.

To make such a comparison we have chosen two security areas, digital signatures and logging, and listed what the ISO/IEC standard for security management specifies. The reason for choosing this standard was that "Health Care 1" and "Highway Tolls" require that standard to be used.

In the case of the "E-Business" system the requirement on digital signatures was formulated as: "The system should be able to handle the use of electronic signatures". Reading the ISO/IEC standard we find detailed information on what to consider when requiring digital signatures:

- Protection of confidentiality of signature keys
- Protection of integrity of public key
- Quality of signature algorithm
- Bit-length of keys
- Signature keys should differ from keys for encryption
- Assure proper legal binding of the signatures

Logging is specified without standards in seven of the studied projects and specified by referral to standards in two of the projects. If we look at the seven projects with no referral to external documents, the ISO/IEC standard again provides requirements left out in the specifications:

- Separation of users logged and reviewers of the log

- Protection against de-activation
- Policy for who can change what to be logged
- Protection against logging media being exhausted

The subcategory “what info to be logged” can be further broken down into specific pieces of information. Three out of the seven projects above have specific requirements in what information to be logged. From the ISO/IEC standard we get the following list of left out requirements:

- User IDs
- Date and time of log-on and log-off
- Terminal ID and location
- Successful and rejected system access attempts and data access attempts
- Archiving of logs

4.4 Possible Shortcomings

There are possible shortcomings to our study. First, we want to stress that we do not have access to any kind of risk analysis documents underlying the security requirements specified. Therefore we cannot know if certain security areas have been left out because of deliberate decisions or because of lack of information or knowledge. As a consequence we do not judge the requirements as good or bad, but rather analyze the consistency and the use of standards.

Some of the requirements found in the specifications studied were hard to categorize in a clear way, mostly due to the diversity in definitions of non-functional requirements. Therefore the categorization should not in all cases be interpreted as a given fact.

Using requirement specifications made for public procurement in Sweden for our field study is a decision made primarily because of the availability of them. Commercial entities tend to have little interest in making their requirement specifications available for research. This limited scope affects the validity of the study.

5 Conclusions

We conclude that current practice in security requirements is poor. Our field study shows that security is mainly treated as a functional aspect composed of security features such as login, backup, and access control. Requirements on how to secure systems through assurance measures are left out. Nonetheless, all systems studied have some form of security requirements and most of them have detailed requirements at least in certain security areas. This shows that security is not neglected as such.

The RE community often conceives security as a non-functional requirement and thus generally hard to manage. Our study shows that security requirements are both functional and non-functional. In the functional case they represent abstract security features broken down into concrete functional requirements. In the non-functional case they are either restrictions on design and implementation, or requirements on assurance measures such as security testing.

Following standards and not relying on local competence would make management of security functional requirements no harder than other functional requirements. Thus security requirements being hard to manage mainly holds for security assurance requirements.

6 Acknowledgments

We would like to sincerely thank the reviewers in the SREIS program committee, and our previewers—David Byers and Kristian Sandahl.

Paper B

The Impact of Neglecting Domain-Specific Security and Privacy Requirements. Published in the Proceedings of the 12th Nordic Workshop on Secure IT Systems (Nordsec 2007), October 11-12, 2007. The layout is modified.

The Impact of Neglecting Domain-Specific Security and Privacy Requirements[1]

John Wilander
Omegapoint AB, and
Dept. of Computer and Information Science
Linköpings universitet

Jens Gustavsson
Dept. of Computer and Information Science
Linköpings universitet

E-mail: `{johwi, jengu}@ida.liu.se`

Abstract

In a previous field study of eleven software projects including e-business, health care and military applications we documented current practice in security requirements. The overall conclusion of the study was that security requirements are poorly and inconsistently specified. However, two important questions remained open; what are the reasons for the inconsistencies, and what is the impact of such poor security requirements? In this paper we seek the answers by performing in-depth interviews with three of the customers from the previous study. The interviews show that mature producers of software (in this case IBM, Cap Gemini, and WM-Data) compensate for poor requirements in areas within their expertise, namely software engineering. But in the case of security and privacy requirements specific to the customer domain, such compensation is not found. In all three cases this has led to security and/or privacy flaws in the systems. Our conclusion is that special focus needs to be put on domain-specific security and privacy needs when eliciting customer requirements.

Keywords: security and privacy requirements, requirements engineering

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

1 Introduction

The security (confidentiality, integrity, availability) and privacy properties of custom-made software relies heavily on the requirements specified by the customers. If the requirements are poorly specified there is no guarantee that the producers of the software will strive for security.

Security and privacy (henceforth grouped as *security* where possible) are often conceived as *non-functional requirements* [70, 71, 72, 73], which are generally hard to manage. But our previous field study on requirements on eleven systems showed that more than 75 % of security requirements found in specifications are in fact functional [83]. We concluded that customers are generally better at specifying functional requirements including functional parts of security. Therefore the hard part of specifying security lies in the truly non-functional aspects such as security processes, security testing, and security evaluation. Section 3 in this paper briefly presents the results of the previous field study.

The outcome of the field study led us to a few hypotheses as to why security requirements are poorly specified, and what the impact of such poor requirements would be. We present these hypotheses in Section 4. To verify the hypotheses we conducted in-depth interviews with customer project leaders from three of the systems in the previous study. Section 5 summarizes the outcome of these interviews. The answers confirmed most of our hypotheses and the discussion can be found in Section 6. In Section 8 we draw the conclusion that customers and producers of software should put special focus on domain-specific privacy needs when eliciting requirements for security critical systems.

2 Terminology

Software systems and stakeholders described in this paper include *customers* that have needs, typically specified as software requirements. The customers buy systems from software *producers* who fulfill requirements and deliver systems. In our frame of reference, customers are not necessarily IT-experts, but rather experts within their own domains such as health care processes or traffic and infrastructure. A few terms we use need to be defined:

Requirement. A requirement is a specification of what the customer needs to be implemented during system development—a description of how the system should behave, or of a system property or attribute [84].

Unspecified need. An unspecified need is a left-out requirement—the customer needs a certain function or system behavior but has so far failed to express this need as a requirement.

Over-delivery. An over-delivery is performed when a producer fulfills more than the customer has explicitly required, typically when the producer fulfills the customer's *unspecified needs*.

Local hero. A local hero is a person with expert knowledge in a subset of a field that is wrongly consulted as an expert in the field in general. An example could be a person with much experience in how to set up and and effectively manage security logging. By others that person could very well be consulted as an expert in system security in general, being the "local security hero".

3 Previous Study

In 2005 we published a field study of current practice covering eleven requirements specifications on IT systems being built 2003–2005 [83]. All specifications were made for public procurement in Sweden, in compliance with EU procurement directives. This choice was made primarily because of the public availability of the specifications. The study covered:

- Five systems for billing, accounting, salary, and e-business
- Three health care systems
- One system for defense materiel
- One system for reporting hazardous materials

- One system for managing highway tolls

Requirements found in the specifications were categorized into security areas and divided into functional and non-functional requirements (for details on how this was done see the original paper [83]). Table 1 contains an overview of all security requirements we found. Note that requirements not present in any of the studied specifications are not in the table, thus no rows with zero requirements. The overall conclusion was that security requirements were poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions.

4 Hypotheses

The outcome of the field study lead us to four general hypotheses about the delivered systems. These were our hypotheses:

4.1 Security Requirements Incomplete

In our previous study we did not have access to any risk analysis documents, nor did we speak with the people involved—we just studied the requirements specifications as such. Therefore we could not know if certain security requirements had been left out because of deliberate decisions or because of lack of information or knowledge. As a consequence we did not judge the requirements specifications as complete or incomplete, but rather analyzed consistency and the use of standards. However, our hypothesis was that the security requirements were indeed incomplete or underspecified.

4.2 Lack of Risk Analysis

In several of the specifications studied we noted that some security requirements were fairly well specified whereas related ones were completely left out. Examples of such inconsistencies could be seen in access control/roles where all systems had requirements indicating that restricted access was important. At the same time only three out of eleven specifications required

Requirements **Systems**	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Access Control/Roles	1	11	6	5	8	5	4	5	3		3
Attack Detection							2	4		3	
Backup		5	9	2			2	2			
Digital Signatures			1		1	1	1	2			1
Encryption							4	1			1
Integration							2	1			
Logging		9	3	1	11	1	5	8	1		
Login		5	3	3	8	2		2	1		2
Privacy			2							1	
Authentication							2	4	2		1
Availability	1		3			1	6	4		3	1
Design/Implementation					1			6			1
Physical Security								6			
Risk Analysis										1	
Security Management								2		2	
Security Testing								1			

Table 1: Overview of previous study—security requirements on eleven IT systems built 2003-2005. The double horizontal line divides the requirement categories into mostly functional (above) and mostly non-functional (below). Numbers tell how many requirements were found in each category.

some kind of encryption of data communication and only two specifications had requirements on restricted physical access.

Our hypothesis was that the specifications had not been preceded by risk analyzes. Such analyzes should have identified a greater variety of threats against important assets, and thus resulted in more consistent requirements.

4.3 Heavy Trust in Local Heroes

Some security requirements had a high level of detail whereas others in the same specification were only specified on a general level. This might indicate that the organizations specifying the security requirements relied heavily on local competence and not standards. Such local competence tends to be strong in certain areas and weak in others. We call this the *local heroes phenomenon*.

Such inconsistent levels of detail could for instance be seen in the "E-Business" system where the requirements on logging were very detailed (eight requirements on what info to be logged) and at the same time digital signatures were specified as "The system should be able to handle the use of electronic signatures" with no further details.

Our hypothesis was that the studied organizations had relied heavily on local heroes.

4.4 Systems Insecure

Considering the three previous hypotheses we arrived at an overall hypothesis that the delivered systems were insecure, and that this had manifested itself as security flaws and insecure operation.

5 Interviews

To verify our hypotheses we conducted interviews with the customers behind three of the requirements specifications, namely Health Care 1, Highway Tolls, and Medical Advice. Before we present the actual interviews we describe our methodology, scope, and potential shortcomings.

5.1 Methodology and Scope

We conducted oral, open-question interviews with the customer project leaders. The interviews lasted 1-2 hours and were recorded and transcribed (approximately 30 pages of text per interview) to allow for an accurate, qualitative analysis. We chose to interview the project leaders since they had a good overview of the requirements process, of the systems and of relations with the producers.

The three systems were chosen specifically because they were all security and privacy critical, they were fairly large, and they represented three interesting categories—a standard system with configuration (Health Care 1), a combination of standard components and development (Highway Tolls), and one system completely built from scratch (Medical Advice). Further, these systems had some of the best security requirements in the previous study (in the case of the Highway Tolls system there were proper references to security standards) which hopefully would work as an upper bound on our analysis, i.e. the other systems were unlikely to show substantially better results when verified against our hypotheses.

The systems were built by WM-data (Swedish company with 9.000 employees, now part of LogicaCMG), IBM, and Cap Gemini. We have chosen not say which company delivered which system, and the customers asked us not to publish man hours or code size since such figures were considered business secrets.

5.2 Potential Shortcomings

We have based our studies on requirements specifications made for public procurement in Sweden—a choice made primarily because of the availability of them. Commercial entities tend to have little interest in making their requirements specifications available for research. This limited scope affects the validity of the study.

A potential problem was the project leaders' technical competence but apart from a few unanswered questions this was never an obstacle. The interview analysis is qualitative and thus subject to the authors' interpretation. All customer quotes presented are translated by the authors.

5.3 Systems

Brief presentations of the systems studied:

Health Care 1 (Customer: Stockholm County Council). Integration platform to support personal medical information following nearly two million patients between various health care organizations. This system is a standard system with only minor new development, and is maintained and run by the producer.

Highway Tolls (Customer: Swedish Road Administration). Equipment, software and services for handling environmental fees for all vehicles entering the city of Stockholm. This system is a combination of standard components and new development, and is maintained and run by the producer.

Medical Advice (Customer: The Federation of County Councils). System for managing medical advice by phone on a national level. It handles redirection of calls, queue management, work-flow management, medical documentation, and statistics. This system was built from scratch for the customer, and is maintained and run by the producer.

5.4 Interview Health Care 1 System

Outcome of the interview with the Health Care 1 project leaders:

Security a critical requirement. The customer considers security to be a critical factor in the system since it contains patient information. A general risk analysis was used to drive parts of the requirements elicitation process, but no specific focus was put on security risks according to what the customer remembers.

Security logs checked by producer. The logs are managed by the producer and any incidents are discussed on a monthly meeting. Since the producer owns the auditing process we asked the customer if they had confidence in the producer telling them of incidents—"We think so. That's a question of conscience!" But what if the producer detects a vulnerability, patches it, but never investigates if the vulnerability was ever exploited? "That's not unlikely. But we're going to hire a security manager that will perform audits of security maintenance."

Security management standard not implemented. The requirements specification referred to the ISO/IEC 17799 standard for security management [77] but the customer admits it has not been implemented yet—"Unfortunately I don't think so."

Vague requirement on separation. Regarding confidentiality the specification stated that "It should be possible to separate different types of information both logically and in terms of security." When asked if they have a clear picture of what was meant the customer replied "No ... Surely, that requirement must have sparked a lot of questions. But I would imagine it refers to privacy categorization of patient information."

Vague requirement on encryption. The health care system was required to "... have functions for protecting the information in the database, for instance through encryption." The customer did not specify what kind of encryption or even that it has to be a standard cryptographic algorithm. So we asked if standard encryption was delivered—"Yes, I think so. They've been talking ... a lot of three letter abbreviations ... It's ongoing. We had a lot of discussions regarding this and it became an issue of negotiation in the end." Here the producer covered up for a poorly specified requirement.

Local heroes phenomenon confirmed. Contrary to the vague specification on encryption the customer had requirements on input and output validation which must be considered being on a low technical level. When asked if this was a manifestation of the local heroes phenomenon they replied "Yes, I think so. We know their names too."

Automatic recovery requirement not fulfilled. The specification contained a requirement on automatic recovery—"The system should automatically handle errors and restart functions and processes." The customer admitted that the requirement was vague, and utterly impossible to fulfill if *all* errors were to be handled automatically. But an incident with a faulty load balancer had revealed that the requirement was not even fulfilled to a basic level. The system had gone down and not restarted.

Need for specific handling of protected identities unfulfilled. After delivery the customer had realized that the system lacked support for handling personal information for patients with protected identities. They considered this a severe flaw. The privacy of such patients introduces a whole new dimension to access control and to date it is not clear if and how such functionality can be introduced.

Summary. The customer feels that the producer of the health care system is mature and has fulfilled most of the fairly well specified security requirements. Vague requirements have been costly both in terms of money and time. The lack of support for handling personal information for patients with protected identities is a clear case of a domain-specific need not specified as a requirement by the customer and not fulfilled by the producer.

5.5 Interview Highway Tolls System

Outcome of the interview with the Highway Tolls project leader:

Security requirements very high. The customer considers the "security requirements very high" for the highway toll system, and thus a risk analysis has been performed both before initial release and during the current development iteration.

Logging features missing. The system logs are continuously checked but "there are deficiencies". The deficiency turned out to be a disability to log what information is *accessed* in the system—only *modifications* are logged. This means that employees could violate both confidentiality (e.g. checking when and where cash transports leave and enter the city) and privacy (e.g. systematically checking people's movements in the city area) without being noticed.

The customer considers this a serious flaw due to an incomplete requirements specification and risk analysis. It was clearly a domain-specific need and if the customer had the chance to re-write the requirements they say they would have hired third-party experts to identify such needs and specify them as requirements.

Security incident despite penetration testing. No penetration testing was explicitly required but was performed by the producer anyway, which is a clear case of over-delivery. The pentest reported an insecurely configured server within the system. But the server was never reconfigured and was later successfully abused in a spam attack. Apart from that the customer does not know of any security incidents, but they are "... not sure the maintainer tells us everything".

Security management standard implemented. The requirements specification referred to the ISO/IEC 17799 standard for security management [77] and the customer feels it has been implemented.

Fraud analysis requirement forgotten. The customer required a holistic risk analysis of potential frauds but does not recall that such an analysis has been done. We asked if they did not know they had required a fraud analysis and customer responded “No, that seems to be the truth”.

Summary. The customer feels that the producer of the highway toll system is mature and has fulfilled some of their unspecified needs, i.e.needs not part of the requirements specification. Despite this, security problems have surfaced (e.g. logging), mostly due to unspecified security and privacy needs understood or noted by the producer.

5.6 Interview Medical Advice System

Outcome of the interview with the Medical Advice project leader:

Security requirements high. The customer considers the security requirements high “since the system handles medical records”. Several laws restrict the handling of such information. Despite this, no risk analysis was performed. Instead the customer relied on in-house experience and competence.

Log analysis process undefined. The specification contains five different requirements on logging. But when asked about processes and routines for checking the logs the customer replied “We don’t know. That aspect is not taken care of.” It might be that the producer checks the logs—“Perhaps. We hope so.” When asked if the producer knows what security and privacy issues to check for in the logs the customer replied “To be honest, I don’t think we’ve discussed such security processes. But I expect the producers to tell us if something happens.”

Protection of logs forgotten. The specifications contained a requirement stating that “... the logs shall be protected against manipulation.” When asked if this requirement has been met the customer replied they “... don’t have a clue”.

Vague logging requirement. The specification said that “... sensitive information shall be logged and protected from manipulation.” What *’sensitive information’* means is never specified. The customer agrees “... that the requirement is vague. It has been made more concrete during project iterations.” But such iterative refinements can “... become a time and money discussion” according to the customer.

No known security incidents. We asked the customer if they have had any security incidents so far—“No, I don’t think so.”

Deliberately no security standard. In the customer’s view it was valid to specify the security requirements without using security standards. One person, admittedly a local hero, was responsible for security—“He has not brought up standards as a possibility. He knows standards and legislations and thus I believe he processed the issue himself and chose not to point toward standards.” But when the lack of requirements on physical security (fire, theft etc.) was brought up, the customer admitted that some security issues have been overlooked. “If we would have had a standard those areas would have been covered.” To integrate standards into the requirements the customer says they would have needed help from a third party.

No security testing. No security tests apart from stress testing of availability were required or performed. “But it would have been nice” the customer commented.

Summary. The customer admits that proper measures to ensure security and privacy in the system have not been taken. Questions regarding processes for continuous log analysis revealed that such aspects had not been thought of before, and that the privacy needs had not been properly specified as requirements. The firm belief in their local hero had obvious disadvantages.

5.7 Results on General Security Requirements

In all cases the customers have had higher security and privacy needs than their requirements specifications reflected. They have relied heavily on their producers to handle technical security. In cases where security requirements were specified they were often vague or incomplete, which had led to negotiations and minor disputes. Also worth noting is that all three customers had security requirements that were either not implemented by the producers or forgotten by themselves. Despite this, the customers were mostly satisfied with the general security of the delivered systems.

5.8 Results on Domain-Specific Security Requirements

In all three cases the customers have had problems with domain-specific privacy concerns:

- Health Care 1: No support for handling patients with protected identities.
- Highway Tolls: No logging of accesses to privacy sensitive vehicle data.
- Medical Advice: No process for checking or protecting security logs although the system handles privacy sensitive patient data.

Part of their privacy needs were not expressed as requirements and were not fulfilled by the producers. Our impression was that the customers had not realized in what ways the systems could viola te privacy until after delivery.

6 Discussion

According to our own experience, effective methods and so called "best practices" for software security are more and more becoming part of general software engineering expertise. Therefore, relying on the producers to deliver security despite vague or left-out requirements is not all bad. Besides, some security requirements can be fulfilled with commercial off-the-shelf products engineered by security specialized vendors, in which case the customer most likely gets much more security features than specified.

In two of the three cases (Health Care 1 and Highway Tolls) the producers delivered more security than required by the specifications. We call this phenomenon *over-delivery*. When asked about over-delivery, the customers said that the producers were "mature" and did not want to deliver an insecure system. But the customers raised doubts as to whether some over-delivered parts, such as documentation and defined processes, were actually considered *intellectual property* of the producers. In the third case

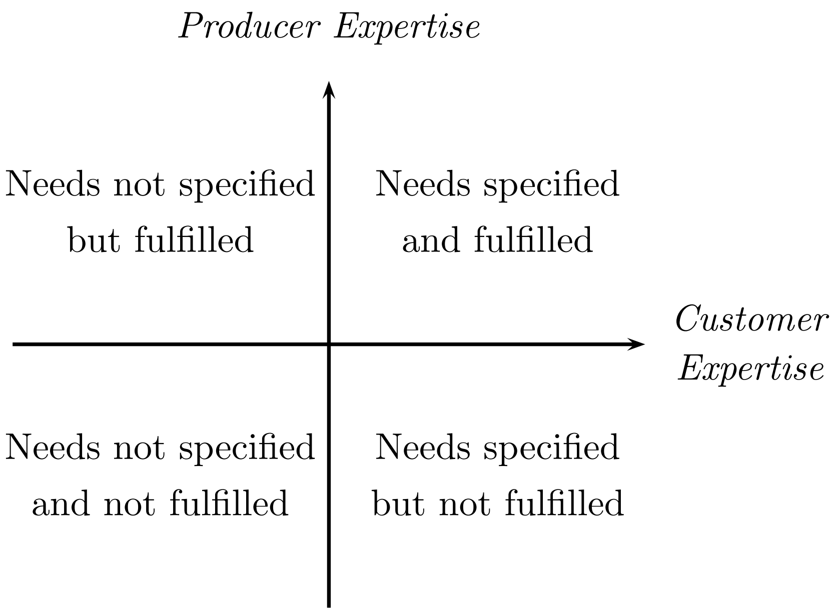

Producer Expertise
Needs not specified but fulfilled
Needs specified and fulfilled
Customer Expertise
Needs not specified and not fulfilled
Needs specified but not fulfilled

Figure 1: Quadrant diagram visualizing the separation between customer and producer expertise, and its consequences for customer needs. When producer and customer understand each other they end up in quadrant one (north east). When a certain need is not specified as a requirement by the customer and not noted by the producer they end up in quadrant three (south west).

(Medical Advice) the customer often had had to re-negotiate to compensate for poor initial requirements.

All three customers had had problems with domain-specific privacy concerns, and the nature of the problems suggests that privacy needs are especially prone to being domain-specific. The customers themselves did not know they had such needs or did not realize in what ways the systems could violate privacy. Contrary to unspecified but more technical security needs, the producers did not over-deliver. A reasonable explanation for this would be that the producers did not know there were such needs within the domains the systems were being built for.

We tried to visualize the separation of domain knowledge and its consequences in terms of fulfilled and unfulfilled needs in a quadrant diagram (see Figure 1). The customers agreed that this was a relevant model of reality and that they had experience from all four quadrants in the projects.

6.1 Verification of Hypotheses

Our hypotheses (see Section 4) were verified against the interview outcome:

- **Security requirements incomplete**. All three customers admitted that their requirements specifications contained vague, inconsistent and incomplete security requirements.
- **Risk analyzes performed**. Two of the systems had performed at least some kind of risk analysis (unspecified which kind). Thus the poorly specified requirements were not clearly due to lack of risk assessment.
- **Local heroes phenomenon confirmed**. All three customers had relied heavily on so called local heroes and could actually name them. Nevertheless, there was an understanding that third-party consultants would probably have mitigated the problems with vague and unspecified needs.
- **Systems partly insecure**. Only one of the systems had had a known security incident, but none of the customers had a defined process for investigating if any security incidents occurred. Two of the systems had serious privacy flaws that had to be fixed in future versions.

6.2 Validation Against Maintainability Requirements

We conducted a parallel study on another non-functional requirement category, namely *maintainability*. The full results of that study is still to be published, but the material allowed us to do a comparative validation of our results on security and privacy requirements.

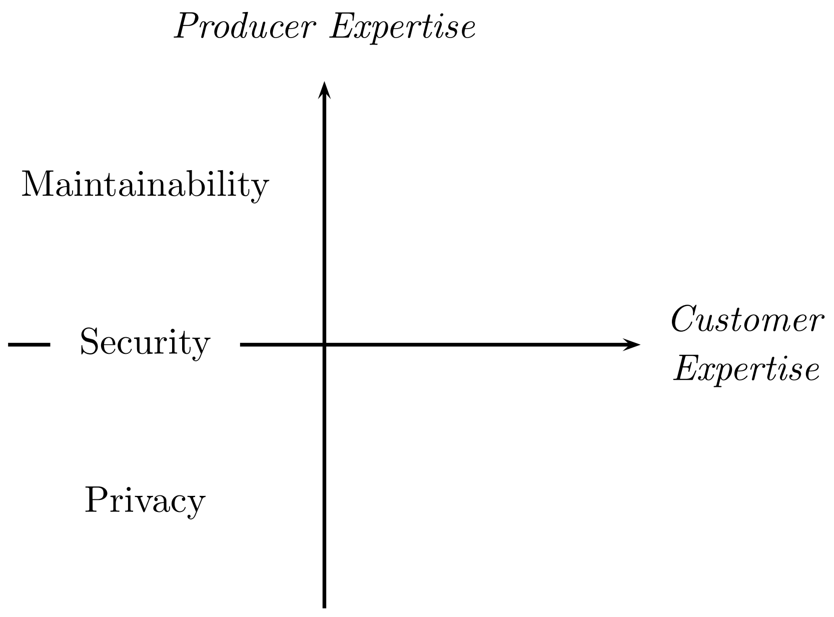


Figure 2: Quadrant diagram visualizing producer expertise within three non-functional requirements categories. Producers typically fulfill unspecified maintainability needs but not unspecified privacy needs.

Compared to security there seems to be a much higher degree of over-delivery in maintainability. One of the reasons for this is that maintainability requirements such as system documentation, regression test suits, and coding standards typically are general, i.e. not specific to the customer domain. Therefore they are part of the producer expertise and is a potential candidate for over-delivery—a mature producer doesn't skip documentation just because the customer failed to require it.

We can place the three non-functional requirements categories in another quadrant diagram (see Figure 2). Maintainability needs are part of general software engineering, (technical) security needs are more and more becoming part of software engineering, whereas privacy needs do not seem to be part of the software engineering domain yet.

7 Related Work

Several research studies have investigated security and privacy requirements (presented below). Unfortunately most of them treat the underlying problem as based on experience or as anecdotal. We hope to fill that gap, but our studies are of course related.

Alderson discusses the fact that vague or underspecified requirements (defined by him as *false requirements*) often express real customer needs [85]. His findings support that requirements specifications often leave out or fail to properly specify customer needs.

Anderson gives an example where a British bank system did not log customer address changes and a clerk abused the system by changing a chosen customer's address to her own, issuing a new ATM card and PIN, and then changing the address back [86]. This was possible since the bank's clerks had privileges to change both customer addresses and issue new ATM cards. A risk analysis would have had to include people with domain-specific knowledge of privileges and procedures at the bank to detect this security threat.

McDermott and Fox note that the security engineering process is complex and hard to understand even for skilled software engineers [87]. As an example they mention theoretical models for RBAC. This supports our conclusion that domain-specific needs will not be covered by software engineers even though they are skilled and have the best intentions.

Alexander shows by example that complex requirements problems can only be solved by the "combined domain knowledge and skill of the stakeholders" [88]. This pinpoints the fact that only the combination of various domain expertises can cover all the customer needs and formulate them as requirements.

Firesmith, Sindre, and Opdahl have shown that security and privacy requirements are often similar or exactly the same across various systems, at least when compared on an abstract, goal-oriented level [89, 90, 91]. This suggests that typical security requirements can be listed and reused in future projects (potentially in a refined form). To prove their point they provide examples of such reusable requirements.

Firesmith et al's results potentially conflict with ours since they seemingly suggest that there are no domain-specific security or privacy requirements. But, as they note themselves, careful elicitation is necessary to be able to *choose* among the reusable requirements. And as mentioned, their reusable requirements are all on an abstract, goal-oriented level, and thus need to be *refined* to comply with the customer's domain-specific needs.

Liu, Yu, Mylopoulos, and Cysnerios have presented a framework for modeling security and privacy requirements using the agent-oriented *i** language [92, 93, 94]. Traditionally, *i** iteratively models *actors*, *actor goals*, and *actor dependencies*. In the context of security and privacy they add models of the counterpart, i.e. *attackers*, *attacker goals*, *vulnerabilities*, and *countermeasures*.

Use cases, introduced by Jacobson [95], have been used to model (mostly functional) requirements for long, for instance within RUP [96]. But security and privacy are often conceived as non-functional requirements [70, 71, 72, 73] and therefore not suited for use cases [97]. To elicit such requirements McDermott and Fox proposed *abuse cases*, which model interactions between systems and one or more actors, where the results of the interactions are harmful to systems, actors, or system stakeholders [87]. Very similar to abuse cases are *misuse cases* [88, 98, 97], *abuse frames* [99], and *anti-requirements* [100]. Since abuse/misuse cases identify and model threats they have much in common with *threat modeling* [101]. But threat modeling involves analysis of data flow and is therefore typically carried out later when there is a high-level design of the system.

8 Conclusions

Current practice in security requirements is in many ways poor. Customers tend to rely on local competence to specify security and privacy requirements, and tend to rely on the software producers to cover up for unspecified requirements.

Mature software producers seem to cover up fairly well for unspecified requirements that are part of the general software engineering domain. But our in-depth interviews with customer representatives show a severe impact of neglecting the specifics of the customer domain in eliciting security and privacy requirements. Unspecified requirements specific to the customer domain are unlikely be fulfilled by software engineers even though the engineers are skilled and have the best intentions.

According to our results, privacy requirements seem especially prone to being domain-specific. All three customers we interviewed had privacy needs that were never specified as requirements and never fulfilled by their producers. Customers need to cooperate with both producers and domain experts such as security specialists to be able to identify their needs and formulate them as requirements. Only the combination of various domain expertises has the potential to cover all the customer needs. This cooperation could be performed within the scope of risk analyzes.

9 Acknowledgments

We would like to sincerely thank the previewers of this paper and of course our interview participants.

Paper C

A Comparison of Publicly Available Tools for Static Intrusion Prevention. Published in the Proceedings of the 7th Nordic Workshop on Secure IT Systems, 2002. The layout is modified.

A Comparison of Publicly Available Tools for Static Intrusion Prevention[1]

John Wilander and Mariam Kamkar

Dept. of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping
Sweden
{johwi, marka}@ida.liu.se

http://www.ida.liu.se/j̃ohwi

Abstract

The size and complexity of today's software systems is growing, increasing the number of bugs and thus the possibility of security vulnerabilities. Two common attacks against such vulnerabilities are buffer overflow and format string attacks. In this paper we implement a testbed of 44 function calls in C to empirically compare five publicly available tools for static analysis aiming to stop these attacks. The results show very high rates of false positives for the tools building on lexical analysis and very low rates of true positives for the tools building on syntactical and semantical analysis.

Keywords: security intrusions, intrusion prevention, static analysis, security testing, buffer overflow, format string attack

1 Introduction

As our software systems are growing larger and more complex the amount of bugs increase. Many of these bugs constitute security vulnerabilities.

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

According to statistics from CERT Coordination Center at Carnegie Mellon University the number of reported security vulnerabilities in software has increased with nearly 500% in two years [102].

Now there is good news and bad news. The good news is that there is lots of information out there on how these security vulnerabilities occur, how the attacks against them work and most importantly how they can be avoided. The bad news is that this information apparently does not lead to less vulnerabilities. The same mistakes are made over and over again which for instance is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows alone stand for about 50% of the vulnerabilities reported by CERT [5]. Equally dangerous is the *format string* vulnerability which was publicly unknown until 2000.

In the middle of January 2002 the discussion about responsibility for security intrusions took an interesting turn. The US National Academies released a prepublication recommending policy-makers to create laws that would hold companies accountable for security breaches resulting from vulnerable products [103] which got global media attention [104, 105]. So far, only the intruder can be charged in court. In the future software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

A good starting point would be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most probably be more effective, but given the amount of existing software, the strive for modular design reusing software components, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools trying to clean up vulnerable source code are necessary. A further discussion on this issue can be found in the January/February 2002 issue of IEEE Software [106].

In this paper we investigate the effectiveness of five publicly available static intrusion prevention tools—namely the security testing tools ITS4, Flawfinder, RATS, Splint and BOON. Our approach has been to first get an in-depth understanding of how buffer overflow and format string attacks work and from this knowledge build up a testbed with identified security bugs. We then make an empirical test with our testbed. This work is a follow-up of John Wilander's Master's Thesis [107].

The rest of the paper is organized as follows. Section 2 describes process memory management in UNIX and how buffer overflow and format string attacks work. Here we define our testbed of 23 vulnerable functions in C. Section 3 presents the concept of intrusion prevention and describes the techniques used in the five analyzed tools. Section 4 presents our empirical comparison of the tools' effectiveness against the previously described vulnerabilities. Related work is presented in section 5. Finally section 6 contains our conclusions.

2 Attacks and Vulnerabilities

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according definitions in for example the Internet Security Glossary [80]. In our context an intrusion or a successful attack aims for *changing the flow of control*, letting the attacker execute arbitrary code. Software security bugs, or *vulnerabilities*, allowing these kind of intrusions are considered the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in his or her new shell, leaving the whole system open for any kind of manipulation.

2.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function writing to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing flow of control is made by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular code pointer to target is the return address on the stack. But programmer defined *function pointers*, so called *longjmp buffers*, and the old *base pointer* are equally effective targets of attack.

2.2 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [5, 108] and has been extensively analyzed and described in several papers and on-line documents [62, 109, 110, 111]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

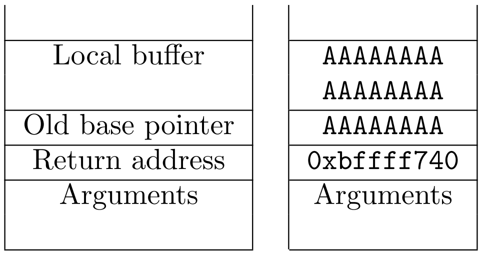


Figure 1: A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

2.3 Buffer Overflow Vulnerabilities

So how come there is no check whether the data fits into the destination buffer? The problem is that several of ANSI C's standard library functions rely on the programmer to do the checking, which they often do not. Many of these functions are powerful for handling strings and thus popular. More secure versions have in some cases been implemented but are not always known by programmers. There are lists of these dangerous C functions often involved in published buffer overflows [112, 113, 69]. From these lists we have chosen to take the fifteen functions considered most risky into our testbed:

1. `gets()`
2. `cuserid()`
3. `scanf()`
4. `fscanf()`
5. `sscanf()`
6. `vscanf()`
7. `vsscanf()`
8. `vfscanf()`
9. `sprintf()`
10. `strcat()`
11. `strcpy()`
12. `streadd()`
13. `strecpy()`
14. `vsprintf()`
15. `strtrns()`

This list is not exhaustive but should provide useful test data for our comparison of the tools.

2.4 Format String Attacks

22nd of June 2000 the first *format string attack* was published [114]. Comments in the exploit source code dates to the 15th of October 1999. Until then this whole category of security bugs was publicly unknown. Since then format string attacks have been acknowledged for being as dangerous as buffer overflow attacks. They are described in an extensive article by Team Teso [115] and also in a shorter article by Tim Newsham [116].

String functions in ANSI C often handle so called *format strings*. They allow for dynamic composition or formatting of strings using *conversion specifications* starting with the character `%` and ending with a conversion specifier. Each conversion specification results in fetching zero or more subsequent arguments.

Let's say a part of a program looks like this:

```
void print_function_1(char *string) {
  printf("%s", string); }
```

A call to `print_func_1()` would print the string argument passed. The same functionality could (seemingly) be achieved with somewhat simpler code:

```
void print_function_2(char *string) {
  printf(string); }
```

Using the function argument `string` directly will still print the argument passed to `print_function_2()`. But what if we call `print_function_2()` with a string containing conversion specifications, for example `print_function_2("%d%d%d%d")`? Then `printf()` will interpret the string as a format string and in this case assume that there are four integers stored on the stack and thus pop four times four bytes of stack memory and print the values stored there. So if programmers take this shortcut when using format string functions, the possibility arises for an attacker to inject conversion specifications that will be evaluated.

Now, considering the conversion specifier `%n` things get dangerous. `%n` will cause the format string function to pop four bytes of the stack and use that value as a memory pointer for storing the number of characters so far in the format string (i.e. the number of characters before `%n`.). So by injecting a format string containing `%n` an attacker can *write* data into the process' memory.

If an attacker is able to provide the format string to a an ANSI C format function in part or as a whole a format string vulnerability is present. By combining the various conversion specifications and making use of the fact that the format string itself is stored on the stack we can view and write on arbitrary memory addresses.

2.5 Format String Vulnerabilities

While the `scanf()`-family is involved in numerous of buffer overflow exploits [117] the format string attacks published concern the `printf()`-family of format string functions [115, 118]. For that reason our test only concerns the latter subset of the ANSI C format functions. So we add another eight function calls to our testbed (`sprintf()` and `vsprintf()` are used differently here than in the buffer overflow case):

16. `printf()`
17. `fprintf()`
18. `sprintf()`
19. `snprintf()`
20. `vprintf()`
21. `vfprintf()`
22. `vsprintf()`
23. `vsnprintf()`

3 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [7] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention.* In this section we will first describe the differences between these two categories. Secondly, we describe five publicly available tools for static intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against vulnerabilities described in section 2.2. This is not a complete survey of static intrusion prevention tools, rather a subset with the following constraints:

- Tools used in the testing phase of the software.
- Tools that require no altering of source code to detect security vulnerabilities.
- Tools that are implemented and publicly available, not system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

3.1 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks. Their general weakness lies in the fact that the protection schemes all depend on how bugs are known to be exploited today, but they do not get rid of the actual bugs. Whenever an attacker has figured out a new attack target reachable with the same security bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs aiming for the same target.

3.2 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [28] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible security targets in the software. Static intrusion prevention removes the attackers tools, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued. In this paper we have chosen to focus on five publicly available tools for static intrusion prevention.

3.3 ITS4

In late 2000 researchers at Reliable Software Technologies, now Cigital, presented a static analysis tool for detecting security vulnerabilities in C and C++ code—*It's the Software Stupid! Security Scanner* or *ITS4* for short [113]. The tool does a lexical analysis building a token stream of the code. Then the tokens are matched with known vulnerable functions in a database. The reason for not performing a deeper analysis with the help of syntactic analysis (parsing) is that such an analysis cannot be made on the fly during programming. ITS4 is built to give developers support while coding, highlighting potential security problems as they are written. Parsing also suffers from being build dependent, not always covering the whole source code because of pre-processor conditionals.

When writing their paper the vulnerability database contained 131 potential vulnerabilities including problems with *race conditions* (not included in this paper, for reference see article by Bishop and Dilger [119]) and buffer overflows. *Pseudo random functions* are also considered risky since they're often used wrongly in security-critical applications. An entry in the database consists of:

- A brief description of the problem
- A high-level description of how to code around the problem.
- A grading of the vulnerability on the scale `NO_RISK`, `LOW_RISK`, `MODERATE_RISK`, `RISKY`, `VERY_RISKY`, `MOST_RISKY`.
- An indication of what type of analysis to perform whenever the function is found.
- Whether or not the function can retrieve input from an external source such as a file or a network connection.

ITS4 has a modular design which allows for integration in various development environments by replacing its front-end or back-end. In fact that was one of the design goals for ITS4. For the moment it only supports integration with GNU Emacs.

The ITS4 security tool is available for download on the Internet.
`http://www.cigital.com/its4/`

3.4 Flawfinder and Rats

Two new security testing tools where released in May 2001—*Flawfinder* developed by David A. Wheeler [120] and *Rough Auditing Tool for Security* (RATS) developed by Secure Software Solutions [121]. They both scan source code on the lexical level, searching for security bugs. Their solutions are very similar to ITS4. When it was noticed that the two teams where developing similar tools they decided on a common release date and on trying to combine the two tools into one in the future.

Just as ITS4 Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, and more. The tool produces a list of potential vulnerabilities sorted by risk. This risk level depends not only on the function, but on the values of the parameters of the function. For example, constant strings are considered less risky than fully variable strings. The Flawfinder 0.19 vulnerability database contains 55 C security bugs.

RATS scans not only C and C++ code but also Perl, PHP and Python source code and flags common security bugs such as buffer overflows and race conditions. Just as Flawfinder and ITS4, RATS has a database of vulnerabilities and sorts found security bugs by risk. The RATS 1.3 vulnerability database contains 102 C security bugs.

Both these security testing tools are invoked from a shell with source code as input. They traverse the code and produce output with risk grading and short descriptions of the potential problems.

The security tools Flawfinder and RATS are available for download on the Internet.

`http://www.dwheeler.com/flawfinder/`

`http://www.securesw.com/rats/`

3.5 Splint

The next static analysis tool we describe is *LCLint* implemented by David Evans et al [122, 123]. The name and some of its functionality originates from a popular static analysis tool for C called *Lint* released in the seventies [124]. LCLint has later been enhanced to search for security specific bugs [28] and the first of January 2002 LCLint got the name *Secure Programming Lint* or *Splint* for short.

The Splint approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis on the syntactic level, making use of the program's parse tree. This means that the tool has a much better chance of differentiating between correct and incorrect use of functions than the tools working on the lexical level.

The annotations specify function constraints in the program—what a function *requires* and *ensures*. Here is a simplified example from the annotated library `standard.h` in the Splint package:

```
char *strcpy (char *s1, char *s2)
      /*@requires maxSet(s1) >= maxRead(s2) @*/
      /*@ensures  maxRead(s1) == maxRead (s2) @*/
```

The `requires` clause specifies that buffer `s1` must be big enough to hold all characters readable from buffer `s2`. The `ensures` clause says that, upon return, the length of buffer `s1` is equal to the length of buffer `s2`. If a program contains a call to `strcpy()` with a destination buffer `s1` smaller than the source buffer `s2`, a buffer overflow vulnerability is present and Splint should report the bug.

To detect bugs the constraints in the annotations have to be resolved. Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by cojoining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if the latter hold. If they do not Splint issues a warning.

Note that we will not add any annotations to our test source code since that would be a violation of the second testing constraint defined in section 3. We rely fully on Splint's annotated libraries to make a fair comparison.

The Splint security tool is available for download on the Internet.

`http://www.splint.org/`

3.6 BOON

David Wagner et al presented a tool in 2000 describing aiming for detecting buffer overflow vulnerabilities in C code [5]. In July 2002 their tool, or rather working prototype, was publicly released under the name *BOON* which stands for *Buffer Overrun detectiON*. Under the assumption that most buffer overflows are in string buffers they model string variables (i.e. the string buffers) as two properties—the allocated size, and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties of the string variable. The constraints are solved and matched to detect inconsistencies similarly to Splint.

Before analyzing the source code you have to use the C preprocessor on it to expand all macros and #include's. Then BOON parses the code and reports any detected vulnerabilities as belonging to one of three categories, namely "Almost certainly a buffer overflow", "Possibly a buffer overflow" and "Slight chance of a buffer overflow". The user needs to go check the source code by hand and see whether it is a real buffer overflow or not. Note that BOON does not detect format string vulnerabilities and is thus not tested for that.

The BOON security tool is available for download on the Internet.
`http://www.cs.berkeley.edu/~daw/boon/`

3.7 Other Static Solutions

There are several other approaches to static intrusion prevention. The area connects to general software testing which provides a broad range of potential methodologies.

A tool yet to be published is *Czech* by Jose Nazario [125]. Czech is a C source code checking tool that will do full out static analysis and variable tainting.

Software Fault Injection

A technique originally used in hardware testing called *fault injection* has also been used to find errors in software [126]. This has been used for security testing. By injecting faults, the system being tested is forced into an anomalous state during execution and the effects on system security is observed and evaluated.

Anup Ghosh et al implemented a prototype tool called *Fault Injection Security Tool*, or FIST for short [127]. The tool shows promising results but preparations of the source code have to be made by hand which means that the process is not automated. Also FIST is not available for download so we have excluded it from our analysis.

Also Wenliang Du and Aditya Mathur have done research on software fault injection for security testing [128]. They inject faults from the environment of the application, i.e. anomalous user input, erroneous environment variables and so on. In their paper they describe a methodology not yet implemented. Therefore their approach is not part of our analysis.

Constraint-Based Testing

Umesh Shankar et al from University of California at Berkeley present an interesting solution to finding format string vulnerabilities [129]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source. Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned of the possible vulnerability. Sadly, we did not manage to get their tool to report any vulnerabilities with the supplied annotated library functions.

4 Comparison of Static Intrusion Prevention Tools

Our testbed contains 20 vulnerable functions chosen from ITS4's vulnerability database (category `RISKY` to `MOST_RISKY`), Secure programming for Linux and UNIX HOWTO [112], and the whole `[fvsn]printf()`-family (see section 2.3 and 2.5 for a complete list). We do not claim that this test suite is perfectly fair, nor complete. But the sources from where we have chosen the vulnerabilities seem reasonable and the test result will at least provide us with an interesting comparison. Our 20 vulnerable functions are used in 13 safe buffer writings, 15 unsafe buffer writings, 8 safe format string calls and 8 unsafe format string calls, in total 44 function calls. We did not go into complex constructs to implement the safe function calls, rather a straight forward solution. An example of the difference between safe and unsafe calls is shown below:

```
char buffer[BUFSIZE];

if(strlen(input_string)<BUFSIZE)
  strcpy(buffer, input_string); /* Safe */
strcpy(buffer, input_string);   /* Unsafe */
```

Overall results from our tests is presented in table 1 and detailed results are presented in table 2. The source code in short form can be found in Appendix A. The exact source code and the print-outs from the various testing tools can be found on our homepage:

`http://www.ida.liu.se/~johwi`

	Flawf.	**ITS4**	**RATS**	**Splint**	**BOON ***
True Positives	22 (96%)	21 (91%)	19 (83%)	7 (30%)	4 (27%)
False Positives	15 (71%)	11 (52%)	14 (67%)	4 (19%)	4 (31%)
True Negatives	6 (29%)	10 (48%)	7 (33%)	17 (81%)	9 (69%)
False Negatives	1 (4%)	2 (9%)	4 (17%)	16 (70%)	11 (73%)

Table 1: Overall effectiveness and accuracy of static intrusion prevention. "Positive" means a warning was issued, "Negative" means no warning was issued. In total 44 function calls, 23 unsafe and 21 safe. * BOON only tested with buffer overflow vulnerabilities.

4.1 Observations and Conclusions

As you would think all three lexical testing tools ITS4, Flawfinder and RATS, perform about the same on the true positive side. After all, a great part of our tested vulnerabilities where found in their databases or in publications connected to them, as stated before. But they differ considerably on the false positives where ITS4 is best.

For security aware programmers with knowledge of how buffer overflow and format string attacks work these tools can be very helpful. They will most probably get minor testing output, be able to sort out what is important and most importantly know how to solve the reported problems. For less experienced programmers the output might be too large and since the tools give no instructions on how to solve the problems they will need some other form of help.

Quite interesting is that Splint and BOON finds so few bugs. We contacted Splint author David Larochelle concerning this and he responded that the undetected bugs where not considered a serious threat since they are known to the security community and easily found with the UNIX command `grep`. We disagree with him—why not detect as many security bugs as possible? And why not help the developers that are not aware of the security vulnerabilities coming from misuse of several C functions?

Splint is the only tool that can distinguish between safe and unsafe calls to `strcat()` and `strcpy()`. This implicates that Splint has a good possibility to accurately detect security bugs with a low rate of false positives,

Vulnerable Function	**Flawf.**		**ITS4**		**RATS**		**Splint**		**BOON**	
	T	F	T	F	T	F	T	F	T	F
gets()	1	-	1	-	1	-	1	-	1	-
scanf()	1	0	1	0	1	1	0	0	0	0
fscanf()	1	0	1	0	1	1	0	0	0	0
sscanf()	1	0	1	0	1	1	0	0	0	0
vscanf()	1	0	1	0	1	1	0	0	0	0
vsscanf()	1	0	1	0	1	1	0	0	0	0
vfscanf()	1	0	1	0	1	1	0	0	0	0
cuserid()	0	-	1	-	1	-	0	-	0	-
sprintf()	1	1	1	0	1	1	0	0	1	1
strcat()	1	1	1	1	1	1	1	0	1	1
strcpy()	1	1	1	1	1	1	1	0	1	1
streadd()	1	1	1	1	1	0	0	0	0	0
strecpy()	1	1	1	1	1	0	0	0	0	0
vsprintf()	1	1	1	0	1	1	1	1	0	0
strtrns()	1	1	1	1	1	0	0	0	0	0
printf()	1	1	1	1	1	1	1	1	-	-
fprintf()	1	1	1	1	1	1	1	1	-	-
sprintf()	1	1	1	1	1	1	1	1	-	-
snprintf()	1	1	1	1	0	0	0	0	-	-
vprintf()	1	1	0	0	0	0	0	0	-	-
vfprintf()	1	1	0	0	0	0	0	0	-	-
vsprintf()	1	1	1	1	1	1	0	0	-	-
vsnprintf()	1	1	1	1	0	0	0	0	-	-

Table 2: Detailed effectiveness and accuracy of intrusion prevention. T = 1 means an unsafe call was found (a true positive), F = 1 means a safe function call was deemed unsafe (a false positive). “-” means no such test is possible.

just as you would think considering its deeper analysis of the code.

The general feeling we get after running the constraint-based testing tools is that they are still in some kind of a prototype state. Splint has been around under the name LCLint for some time and is used for general syntactical and semantical testing. But the security part needs to be completed. BOON is published as a prototype and should of course be judged as such.

None of the tools has high enough true positives combined with low enough false positives. Our conclusion is that none of them can really give the programmer peace of mind. And combining their output would be tedious.

5 Related Work

We have found one comparative study made of static intrusion prevention tools—”Source Code Scanners for Better Code” [130] by Jose Nazario. He compares the result from ITS4, Flawfinder and RATS when testing a part of the source code for OpenLDAP known to be vulnerable. It only contains one call to one of our 23 vulnerable functions—`vsprintf()`. No test for false positives is done either.

A study with another focus but relating to ours is “A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services” by Broadwell and Ong [131]. They investigate the strengths of static analysis versus software fault injection in finding errors in several large software packages such as Apache and MySQL. In static analysis they use ITS4 to find race conditions and BOON to find buffer overflows.

6 Conclusions

We have shown that the current state of static intrusion prevention tools is not satisfying. Tools building on lexical analysis produce too many false positives leading to manual work, and tools building on deeper analysis on syntactical and semantical level produce too many false negatives leading to security risks. Thus the main usage for these tools would be as support during development and code auditing, not as a substitute for manual debugging and testing.

Paper D

Modeling and Visualizing Security Properties of Code using Dependence Graphs. Published in the Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden, 2005. The layout is modified.

Errata. During the defense of John Wilander's licentiate thesis the opponent Dr. Andrei Sabelfeld pointed out an erroneous statement in this paper. In Section 4.1 it is stated that "... correct input validation is both a liveness property (...) and a safety property." This is not true. Correct input validation is solely a safety property since execution can be halted before a sensitive use of unvalidated input.

Modeling and Visualizing Security Properties of Code using Dependence Graphs[1]

John Wilander
Dept. of Computer and Information Science
Linköpings universitet
johwi@ida.liu.se

Abstract

In this paper we discuss the problem of modeling security properties, including what we call the *dual modeling problem*, and ranking of potential vulnerabilities. The discussion is based on the results of a brief survey of eight existing static analysis tools and our own experience. We propose dependence graphs decorated with type and range information as a generic way of modeling security properties of code. These models can be used to characterize both good and bad programming practice as shown by our examples. They can also be used to visually explain code properties to the programmer. Finally, they can be used for pattern matching in static security analysis of code.

Keywords: security properties; dependence graphs; static analysis

1 Introduction

According to statistics from CERT Coordination Center, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software [67]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

approximately \$666 million in 2003 [68]. One way of countermeasuring these problems is using security tools to find the vulnerabilities already during software development.

In recent years a lot of research has been done in the field of static analysis for security testing. This research has resulted in several tools and prototypes based on various techniques, models and user involvement. Some of them are publicly available, some are not.

In November 2002 we published a comparative study of five tools publicly available at the time [132]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed. Although the research behind these tools and prototypes is often excellent and the empirical results are promising, it is not evident if and how the techniques can be combined to solve several security problems at once. They all focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user or programmer involvement. Further, to our knowledge there is no thorough study of the problems in modeling security properties that underlie static analysis.

1.1 Paper Overview

In Section 2 we present related work by doing a brief survey of eight existing static analysis tools performing syntactical and semantical static analysis to check security properties. A summary defines the problems we want to solve.

Graph models of security properties in code as a mean for visual communication with programmers is discussed in Section 3. Section 4 provides a definition and discussion of the *dual modeling problem* in the context of security properties in code. Criteria for severity ranking of security vulnerabilities are listed in Section 5.

In Section 6 we propose a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are explained in Section 7 and serve as examples of how the modeling formalism can be used.

Sections 8 and 9 discuss future work and provide our conclusions.

2 Survey of Static Analysis Tools

Static analysis tools try to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them.

The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem. This paper tries to address these two drawbacks by proposing a way to model security properties of code that allows for both effective static analysis and visual communication with the programmer.

Several tools perform a deep analysis on a syntactical and semantical level. We have found eight such tools, all analyzing C code—Splint, BOON, CQual, Metal/xgcc, MOPS, IPSSA, Mjolnir, and Eau Claire. As some of these tools are still being developed and some are not even available as prototypes we do not know to what extent they are used in practice.

2.1 Splint

Secure Programming Lint or *Splint* was implemented by David Larochelle and David Evans [28].

Their approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis, making use of the program's parse tree. The annotations specify function constraints in the program—what a function *requires* and *ensures*.

Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by co-joining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if

Table 1: Overview of static analysis tools checking C code for various security properties (cont.). “Intra” and “Inter” = intra- or interprocedural analysis, “Alias” = data aliasing, “Ptr” = pointer analysis, “Type” = type and type conversion information, and “Annot” = code annotations.

	Control-flow		Data-flow					Annot
Tool	Intra	Inter	Intra	Inter	Alias	Ptr	Type	
Splint	x		x		x			x
BOON			x	x				
Cqual			x		x		x	x
MOPS	x	x						
Metal/ xgcc	x	x	x	x				
IPSSA	x	x	x	x	x	x	x	
Mjolnir	x	x	x	x	x			
Eau Claire	x	x	x	x				

the latter hold. Splint only performs intraprocedural data-flow analysis, and the control-flow analysis is limited.

2.2 BOON

David Wagner et al presented *Buffer Overrun detectiON*, or *BOON*, aiming for detection of buffer overflow vulnerabilities [5]. In July 2002 a prototype was publicly released under the name . Under the assumption that most buffer overflows are in string buffers they model string variables (i.e. the string buffers) as abstract data types consisting of the allocated size and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties. Analysis is carried out by solving integer range constraints.

BOON reports any detected vulnerabilities as belonging to one of three categories, namely “Almost certainly a buffer overflow”, “Possibly a buffer overflow” and “Slight chance of a buffer overflow”.

2.3 Cqual

The tool *Cqual* uses constraint-based type inference [133]. It traverses the program's abstract syntax tree and generates constraints that capture the relations between type qualifiers. A solution to the constraints gives a valid assignment of type qualifiers to the variables in the program. If the constraints have no solution, then there is a potential bug.

Umesh Shankar et al have used Cqual to find format string vulnerabilities [129]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source (Cqual requires the user to manually tag untrustworthy data sources). Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned.

The same tainted functionality was used by Chen et al to statically find implicit type cast errors constituting security vulnerabilities [134]. Johnson and Wagner are using Cqual to check for insecure pointer handling between kernel and user-space in Linux [135].

2.4 Metal and xgcc

Ashcraft and Engler have done security research in the area of meta-level compilation. With their compiler extension *xgcc* and extension language *Metal* they have statically analyzed code for input validation errors on integer variables [136]. C programs are modeled as control-flow graphs and are analyzed path by path.

By formulating rules in Metal they check that integer values coming from untrusted sources are bounds checked before they are used in any sensitive function. The security bugs found are unvalidated integers used in pointer arithmetic, and integer overflows. Memory management errors (`malloc()`/`free()`) were also found but not substantially analyzed.

Potential bugs found are ranked by properties such as local vs global scope, distance in lines of code, and non-aliased vs aliased variables.

2.5 MOPS

Chen and Wagner have designed a static analysis tool called MOPS which checks ordering constraints [51]. Some security bugs can be described in terms of temporal safety properties. MOPS specifically checks dropping of privileges and race conditions in file accesses. C programs are modeled as *push-down automata*, and the security properties are modeled as *finite state automata*. Security models can be combined into complex security properties.

No data-flow, pointer, or aliasing analysis is done, which is justifiable since only temporal properties are checked.

2.6 IPSSA

Livshits and Lam have defined and used an extended intermediate form for finding buffer overflow and format string bugs [137]. Their program model builds on *static single assignment* (*SSA*) form—an intermediate code representation that separates values operated on from the locations they are stored in which is very useful in for instance optimization [138]. The extension, called *IPSSA*, provides interprocedural definition-use information with indirect memory accesses via pointers. It can then be used to perform static analysis that handles pointer and aliasing analysis. Security properties are modeled using a "small special-purpose language designed for the purpose". While technical details of this special-purpose language are lacking their empirical results are very promising, especially the low rate of false positives. Their solution was chosen to be unsound for scalability reasons.

2.7 Mjolnir

Weber et al have presented a tool called *Mjolnir* which makes use of dependence graphs and constraint solving to find buffer overflows in C code [139]. They represent buffers with the same range variables used in BOON (see Section 2.2), build system dependence graphs, decorate them with range constraints based on the semantics of C string library functions, and finally solve the constraint sets.

Table 2: Overview of static analysis tools checking C code for various security properties. The program models are control-flow graph (CFG), abstract syntax tree (AST), push-down automata (PDA), parse tree (PST), static single assignment (SSA), system dependence graph (SDG), guarded commands (GC).

	Program model						
Tool	CFG	AST	PDA	PST	SSA	SDG	GC
Splint	x						
BOON				x			
Cqual		x					
MOPS			x				
Metal/xgcc	x						
IPSSA					x		
Mjolnir						x	
Eau Claire							x

To decorate the dependence graphs they traverse the program bottom-up and generate summary nodes containing the constraints of the current function and all its callees.

Weber et al do not clearly state how safety constraints are generated, but we assume they generate them only for statically allocated buffers. They provide both control-flow insensitive and control-flow sensitive constraint generation. Although global variables normally are handled in dependence graphs (see Section 6.1) they are not handled by Mjolnir. No pointer analysis is done.

2.8 Eau Claire

In spring 2002 Brian Chess presented his tool *Eau Claire* [140]. The tool translates C code into so called *guarded commands*, enhanced with exceptions, assertions, assume statements, and erroneous states. Vulnerabilities are modeled using the ESC/Modula2 specification language where you define what a function requires, modifies, and ensures.

Table 3: (Continued) Overview of static analysis tools checking C code for various security properties. The property models are constraint based (CB), finite state automata (FSA), "Metal" (MET), ESC/Modula2 specification language (ESC), and other, special purpose modeling (OTH).

	Security property model				
Tool	CSB	FSA	MET	ESC	OTH
Splint	x				
BOON	x				
Cqual	x				
MOPS		x			
Metal/xgcc			x		
IPSSA					x
Mjolnir	x				
Eau Claire				x	

Eau Claire then augments guarded commands with the specifications. The outcome is a set of verification conditions which are processed by an automatic theorem prover to find potential violations.

Shortcomings of Eau Claire's static analysis are the conservative approach to pointer dereferences (it assumes that any two pointers of the same type may reference the same location) and references into structures and unions. Type-based vulnerabilities are not targeted by Eau Claire [141].

2.9 Summary

Tables 1, 2, and 3 summarize the properties and features of the tools above.

We conclude that several categories of security properties can be statically checked but there is need of a generic solution. The first step toward such a solution is to define a modeling formalism that both covers all necessary aspects and allows for static analysis.

Two other key issues are that such a solution has to allow for effective feedback to the programmers who have to fix the security problems, and it has to support intuitive modeling of new security properties for effective updates of the database. None of the tools presented above have any other kind of input or feedback than text.

We require that the modeling formalism can:

- visually communicate with programmers who model or fix security problems in code (Section 3);
- model several types of security properties (Section 4);
- rank the severity of potential flaws (Section 5); and
- take into account data-flow, control-flow, type and range information, and combinations thereof (Section 6).

3 The Need for Visual Models

As mentioned in Section 2 the two main drawbacks of static analysis tools are that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem.

Current tools such as the ones briefly presented in Section 2 use textual models of security properties in their databases to give textual feedback to the user. For example Splint gives output in the following manner:

```
bounds.c:9: Possible out-of-bounds store:
  strcpy(str, tmp)
  Unable to resolve constraint:
  requires maxSet(str @ bounds.c:9) >=
  maxRead(getenv("MYENV") @ bounds.c:7)
   needed to satisfy precondition:
  requires maxSet(str @ bounds.c:9) >=
  maxRead(tmp @ bounds.c:9)
   derived from strcpy precondition: requires
  maxSet(<parameter 1>) >=
  maxRead(<parameter 2>)
```

Just as call graphs and and flow graphs can help programmers understand code in general (Grammatech's tool "CodeSurfer" is a perfect example [12]), visual models and graph representations of security properties can help to understand and fix security flaws. Especially when the flaws include interprocedural data- and control-flow dependencies.

4 The Dual Modeling Problem

A common issue in security modeling is what we call the *dual modeling problem*—the problem of modeling malicious or benign things. When modeling security properties of code we need both kinds—models of bad programming practice, and models of good programming practice.

In a seminal paper from 1977 Leslie Lamport describes a formalism closely related to the dual modeling problem—a property stating that nothing bad happens during execution is called a *safety property*, and a property stating that something good (eventually) happens during execution is called a *liveness property* [142].

Typical for a safety property is that we can detect a property violation between one execution step and another. During execution we can look ahead and see if the next execution step will take us into a bad state and in such a case raise an alarm or terminate execution. All run-time security measures such as intrusion detection systems and anti-virus applications detect safety properties—they either try to match with known bad behavior, or they monitor for deviations from good behavior.

In the case of a liveness property we can only detect property violations at termination since during execution, we never know whether the good thing will eventually happen or not. Fulfilling the liveness property could potentially be the last execution step before termination. Therefore we cannot rely on run-time monitoring to countermeasure security vulnerabilities that are violations of liveness properties. Static methods, on the contrary, can look into the "future" by following possible execution paths all the way to termination, and try check if a program satisfies a liveness property.

However, models of good or bad programming practice do not correspond directly to safety and liveness properties. Instead they can be a combination of safety and liveness as explained in Section 4.1 and 4.2.

A comprehensive discussion on this fundamental difference between safety and liveness security properties can be found in Schneider's paper

"Enforceable Security Policies" [143].

4.1 Modeling Good Security Properties

Some security properties of code are typically described as "If you do A you must do B". These properties are best modeled as good programming practice—"do like this".

An example is *input validation* of integers. When an integer can be affected by input from users, files, the network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

While being a model of good programming practice correct input validation is both a liveness property (external input must eventually be validated assuming it will be used sometime), and a safety property (no sensitive use of external input without validation).

4.2 Modeling Bad Security Properties

Some security problems are typically described as "If you do A then you must not do B". Such properties are best modeled as bad programming practice—"do not do like this".

An example of such a problem is the double `free()` vulnerability. Freeing the same memory chunk twice or more may open up for heap corruption attacks.

Trying to model all possible benign ways of freeing memory is infeasible since that would be the same as building complete models of all well-behaved programs using `free()`. A model of a bug, however, covers all cases. The absence of multiple `free()` is a safety property.

5 Ranking of Potential Vulnerabilities

Engler and Musuvathi have clearly pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—“It’s not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ...” [144]. Based on our knowledge and experience on static analysis we propose using the following information from the analysis to generate severity ranking:

- Pointer analysis is a hard problem to solve accurately and thus the risk for false positives increases with the amount of such analysis. Therefore we propose that the more pointer analysis involved in finding a flaw, the lower the ranking.
- Aliasing is another problem in static analysis. Because of potential inaccuracy in the analysis we therefore propose that the more aliasing involved in finding a flaw, the lower the ranking.
- Interprocedural control-flow may result in infeasible execution paths being analyzed. Again, because of potential inaccuracy in the analysis, flaws involving interprocedural analysis are ranked lower than intraprocedural ones.
- Flaws involving implicit events are ranked higher than explicit ones since implicity imposes a higher risk for unintended behavior. An example of this is implicit versus explicit type-casts.

5.1 Using the Dual Model for Ranking

In some cases we can make use of modeling both good and bad programming practice. If we have reached a concise description of a property in one distinct model, the dual of that model often explodes into several cases.

For instance, in the case of implicit type-casting and integer signedness vulnerabilities a model of good programming practice is to validate the integer and to have no implicit type-casts at any use points (this example is explained in detailed in Section 7.1).

The dual of this model contains several ways of violating the property. Various narrowing type-casts and missing validation points can be combined. The benefit of exploding the dual and creating all these models is that we can possibly rank them in terms of severity. Perhaps a certain violation is definitely a security vulnerability, whereas another violation only *might* be vulnerable.

6 A More Generic Modeling Formalism

To meet the requirements listed in Section 2.9 we propose decorated dependence graphs as a more generic formalism for visualizing and modeling security properties, and performing static analysis. We here present intraprocedural and interprocedural dependence graphs, decorated with range and type information. We end the section with a view on possible analysis techniques.

6.1 Program Dependence Graphs

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [145]. While originally generated for procedural languages such as C, algorithms generating dependence graphs for object oriented languages exist, e.g. Java [146]

A dependence graph is an intermediate representation of code where vertices represent statements and predicates (henceforth called *program points*), and edges represent control- and data-flow *dependence*. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph.

A program point B is *control dependent* on another program point A, if A controls whether B is executed or not. Formally A is the first program point not *post-dominated* by B when traversing the control-flow graph backward from B. Informally we can say that program point A is a conditional and B is executed in only one of A's outgoing paths.

A program point B is *data dependent* on a program point A if some variable x is defined in A and later used in B without any new defines in-between. Data dependence can also be in form of definition order. Figure 1 shows a small C function with its corresponding program dependence graph.

6.2 System Dependence Graphs

The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [147]. To generate the SDG we need to encode

```
void func() {
  int sum=0, i=1;
  while(i<11) {
    sum=sum+i;
    i=i+1; }
  printf("%d\n",sum);
  print("%d\n",i); }
```

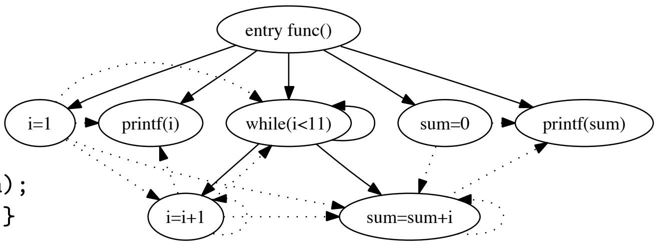


Figure 1: A small C function (left) with its corresponding program dependence graph (right). Solid arrows represent control-flow dependence, dotted arrows represent data-flow dependence. All dependencies are transitive (if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$).

data- and control-flow dependence between procedures which includes formal and actual parameters, formal and actual return values, and global variables.

A procedure call from procedure A to procedure B is modeled with a call vertex in A, an entry vertex in B, and an interprocedural control dependence edge between them. Parameters are handled with actual-in and actual-out vertices in A, formal-in and formal-out vertices in B, and interprocedural data dependence edges connecting them. Temporary variables are used for parameter passing by value-result.

If a procedure uses a global variable, it is treated as a (hidden) input parameter, and is encoded as additional actual-in and formal-in vertices. For further information on summary edges for avoiding calling context problems see the original paper [147].

6.3 Range Constraints in SDGs

Weber *et al* have used decorated SDGs to statically detect buffer overflow vulnerabilities [139]. The graph is augmented with range constraint information for string buffers. Each PDG contains a summary vertex with range constraints of the procedure and all its callees.

```
void copy(char *src) {
  char dst[10];
  strcpy(dst, src); }
```

The PDG for the code to the left would have a range constraint node summary node saying Len(`src`) $\subseteq$ Len(`dst`).

6.4 Type Information in SDGs

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [134].

We propose that the original SDGs be decorated with type information, specifically implicit type conversions. Type conversion information should belong to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others. See Figure 5 and 5 for examples of this decoration.

6.5 Static Analysis Using SDGs

Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* and *chopping* and are used for optimization [148].

A program slice is the parts of a program that can affect the value of a chosen program point, the *slicing criterion.*

Static slicing, invented by Weiser [149], was defined as a reachability problem in PDGs by Ottenstein and Ottenstein [145]. Interprocedural slices can be computed in a similar way in SDGs.

The combination of two (or more) program points, potentially a point with (malicious) user input, and a point with a vulnerability, allows for program chopping—a technique presented by Reps *et al* [150]. When chopping we want to know how some source points affect some target points.

Slices and chops of programs can help with understanding the cause of a vulnerability since they show exactly what parts of the program affect the execution of the vulnerable program point. The richness of program information found in SDGs together with slicing, chopping, type inference and range analysis means it covers all the features of the tools surveyed

```
void func1(char *dest, char *src,
           int len) {
  if(len<MAX)
    memcpy(dest, scr, len); }
```

Figure 2: Implicit type-cast flaw (`len` casted to unsigned int in the call to `memcpy()`).

```
void func2(unsigned int size) {
  char *buf =
       (char *) malloc(size+1);
}
```

Figure 3: Integer overflow flaw (adding one to `size` may cause overflow).

in Section 2 and provides visual communication with the user via a graph representation of the original code.

7 Modeling Security Properties

In this section we show how four security properties can be modeled in terms of decorated dependence graphs. We show the use of dual models both for benign and malicious properties, and ranking of potential flaws. Our proposed formalism is not limited to these properties; they simply serve as examples.

In the graphs all edges represent interprocedural transitive dependence—solid arrows for control-flow, and dotted arrows for data-flow.

7.1 Integer Flaws

Handling integers may seem harmless and straight forward. But several security vulnerabilities prove this a difficult area. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and when the integer representation changes from signed to unsigned or vice versa. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing type-casts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

We are now able to encode the first correct code pattern in terms of our decorated dependence graphs (see Fig. 4). The nodes are program points where "ext input" means external input, "def" means a variable is defined, "val" means a variable is validated, and "use" means a variable is used. The input has to be validated before it is used which means that the use point has to be control dependent on the validation point.
Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [136].

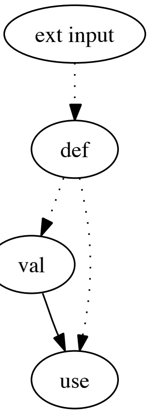


Figure 4: Correct code pattern for integer input validation.

Deviations from this good programming practice, i.e. integer security bugs, have been studied by Blexim [151], Howard [152], and Ashcraft and Engler [136] and we here briefly present the bug types they have identified:

Integer Signedness Errors.

Integer signedness errors can arise both due to implicit type-casting and insufficient validation. In Fig. 2 the signed integer `len` can be negative and as such pass the (inadequate) validation point. When calling `memcpy()` an implicit narrowing type-cast to `size_t` (unsigned integer) occurs which will convert a negative integer to a huge positive integer, possibly overflowing the destination buffer `dest`.

Integer Overflow/Underflow.

When an unsigned integer has reached the maximum value it can represent, an increment to that integer will make it wrap around and become zero. Decrementing an unsigned integer below zero will result in the maximum value.

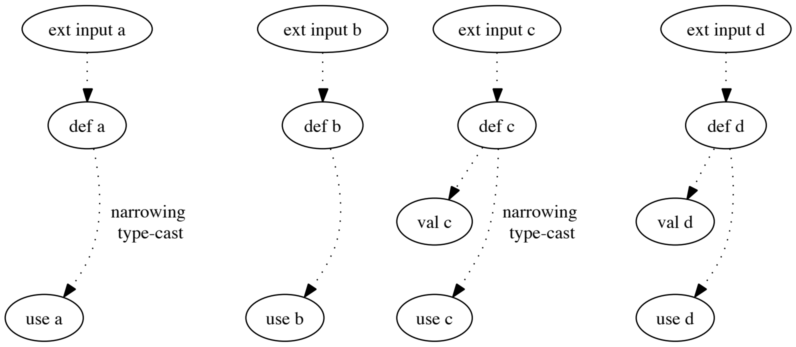


Figure 5: Four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). The proposed severity ranking from left to right is explained in Section 7.2.

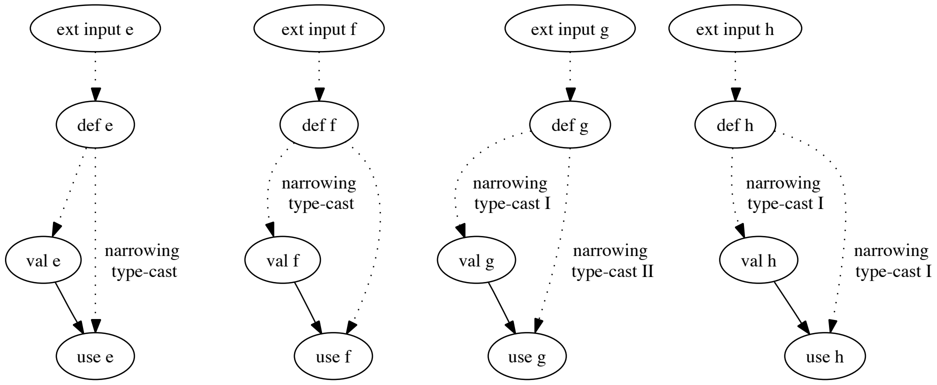


Figure 6: (Cont.) Four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). “narrowing type-cast I” and “narrowing type-cast II” means two *different* type-casts. The proposed severity ranking from left to right is explained in Section 7.2.

In Fig. 3 the intent is to allocate the requested memory plus space for a null terminator. If `size` was the maximum unsigned integer possible, adding one will make it wrap around and call `malloc()` with zero as argument. The return value in such a case is either a null pointer or a non-null pointer that must not be used. Dereferencing such a non-null pointer may allow for heap corruption.

Integer Input Validation.

When an integer can be affected by input from users, files, network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

7.2 Modeling Integer Flaws

To allow for severity ranking we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 5 and 6). Using the ranking rule for implicity (see Section 5) we rank the incorrect code patterns in descending order as follows:

1. Missing validation and narrowing type-cast
2. Missing validation but no narrowing type-cast
3. Use not control dependent on validation and narrowing type-cast
4. Use not control dependent on validation but no narrowing type-cast
5. Narrowing type-cast on either validation or use (two graphs in Fig. 6)
6. Different narrowing type-casts on validation and use
7. Same narrowing type-casts on validation and use

```
char *buf = (char *) malloc(SIZE);
...
free(buf);
...
free(buf);
```

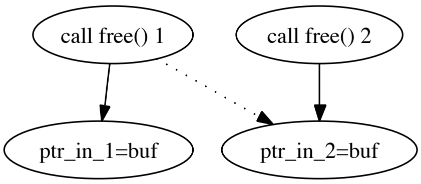


Figure 7: Incorrect code pattern for `free()` and the corresponding dependence graph. If there had been a new call to `malloc()` in-between the two calls to `free()` there would not have been a data dependency edge between the first call to `free()` and the second pointer to `buf` in the graph.

7.3 The Double `free()` Flaw

Often "normal" bugs turn out to be tools for attackers. This is the case of *double free*. To allocate heap memory, the program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`.

To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored "in-band". When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault*. But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of a model of a bad security property (see Fig. 7). We show in Fig. 8 and 9 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`; we also have to look for bad use of `free()`.

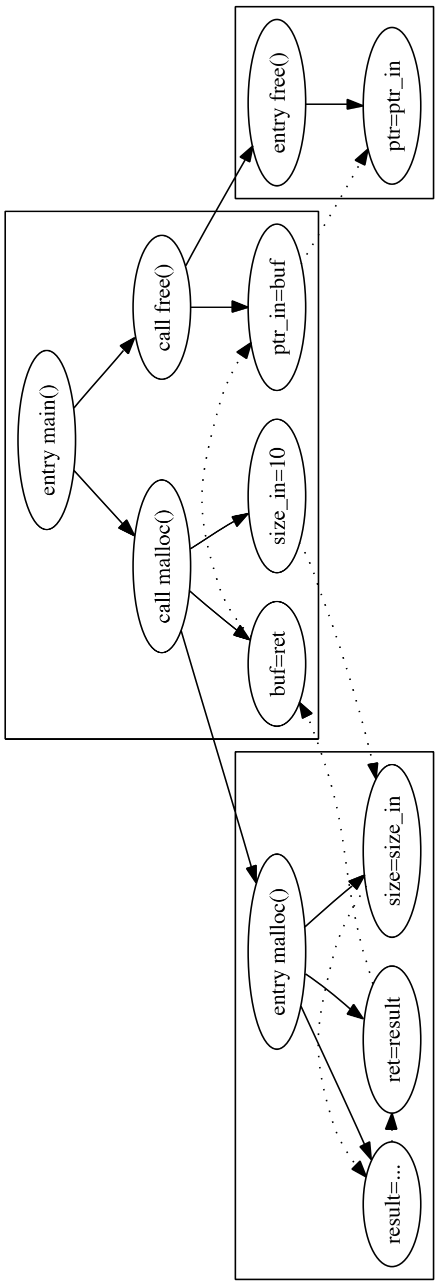


Figure 8: Correct graph pattern for `malloc()` and `free()`.

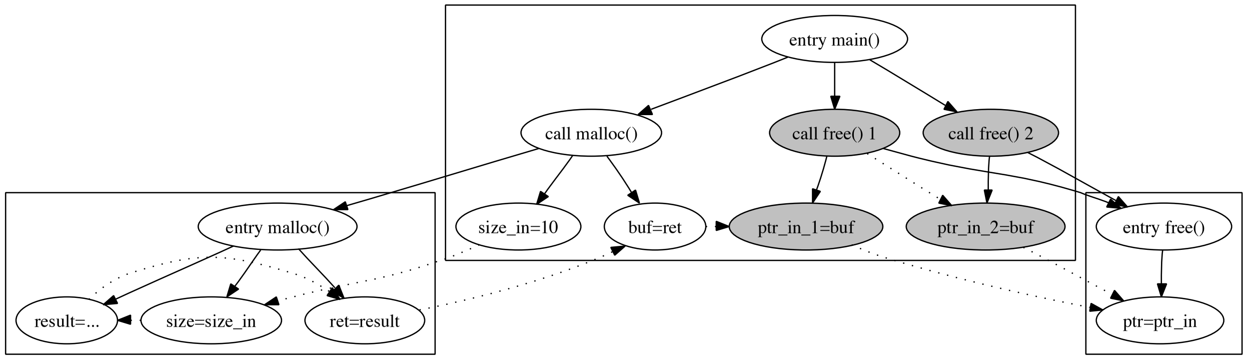


Figure 9: Incorrect graph pattern for `malloc()` and `free()`, where `free()` is called twice. Notice how the grey nodes in the `main()` box match the incorrect code pattern for `free()` which was shown in Fig. 7.

7.4 Modeling External Input

Knowing which data sources not to trust is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. *Environment variables* are considered untrustworthy sources [153], and Ashcraft and Engler add another three categories—System calls, routines that copy data from user space, and network data [136]. In modeling security properties these sources of so called *tainted* data will all be considered as nodes of external input and analyzed via transitive data dependencies.

8 Future Work

Finding the modeling formalism is the first step toward a single tool able to check for several security properties. We are right now implementing a prototype tool called *GraphMatch* that uses dependence graphs to check security properties [13]. The prototype currently finds interprocedural input validation flaws. Apart from modeling other security properties and checking them with real-life code, we plan to investigate scalability and accuracy issues of the analysis, and also evaluate dependency graphs as a visual aid in secure programming. Empirical studies will be made to evaluate the heuristic ranking of potential vulnerabilities.

9 Conclusions

We have shown that there is a need for a generic formalism both for description of security properties and for static checking of these properties. In addition we believe that visual support is needed to effectively communicate with programmers. System dependence graphs decorated with range constraints and type conversion information can serve that purpose. Dependence graphs are well-known in the static analysis and compiler communities and are able to model the diversity of security properties, covering both safety and liveness properties of code, as shown by our examples.

10 Acknowledgments

We would like to sincerely thank the previewers of this paper, especially David Byers.

Paper E

Pattern Matching Security Properties of Code using Dependence Graphs. Published in the online Proceedings of the 1st International Workshop on Code Based Software Security Assessments (CoBaSSA 2005), November 7, 2005. The layout is modified.

Pattern Matching Security Properties of Code using Dependence Graphs[1]

John Wilander and Pia Fåk, {johwi, x05piafa}@ida.liu.se
Dept. of Computer and Information Science, Linköpings universitet

Abstract

In recent years researchers have presented several tools for statically checking security properties of C code. But they all (currently) focus on one or two categories of security properties each. We have proposed dependence graphs decorated with type-cast and range information as a more generic formalism allowing both for visual communication with the programmer and static analysis checking several security properties at once. Our prototype tool *GraphMatch* currently checks code for input validation flaws. But several research questions are still open. Most importantly we need to address the complexity of our algorithm for pattern matching graphs, the accuracy of our security models, and the generality of our formalism. Other questions regard the impact of security property visualization and heuristics for ranking of potential flaws found.

Keywords: security properties; dependence graphs; static analysis

1 Introduction

In November 2002 we published a comparative study of five static analysis tools checking C code for buffer overflows and format string vulnerabilities [132]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed [136, 51, 140, 137, 133, 139]. The research behind these tools and prototypes is excellent and the empirical results are promising, but it is not evident if and how the techniques can be combined to solve several security problems at once. They all (currently) focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user involvement. In our studies of the modeling formalisms used in the tools we identified a specific problem in modeling security properties of code—*the dual modeling problem.*

Some security problems are typically described as “If you do A you must do B” (e.g. *input validation*). Such properties are best modeled as good programming practice—“do like this”. Other security problems are described as “If you do A then you must not do B” (e.g. *double free*). Such properties are best modeled as bad programming practice—“do not do like this”. For a formalism to be able to cover the great variety of security properties it needs to be able to model both good and bad programming practice. The dual modeling problem is closely related to *safety* and *liveness properties* of code [142].

A drawback of static analysis tools in general is that they only *detect* vulnerabilities and therefore the user has to know how to patch the code. The aforementioned tools only offer textual information about analysis results. We believe visual information can be helpful for programmers.

Engler and Musuvathi have pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—“It’s not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ...” [144]. Therefore we believe it is necessary to automatically *rank* the bugs reported from a security analysis tool.

Our research goal is to implement a tool that can:

- check several types of security properties;
- visually communicate with programmers; and

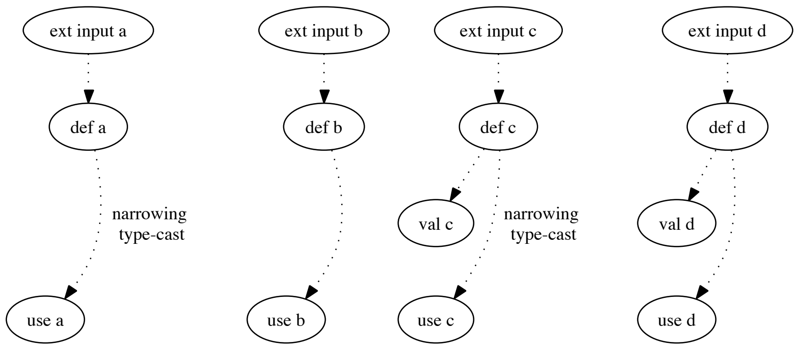


Figure 1: First four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). Severity ranking from left to right stretching from validation point absent to validation but execution not necessarily passing the validation node.

- rank the severity of potential flaws.

1.1 Paper Overview

In Section 2 we present decorated *dependence graphs* as a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are shown in Section 3 and 4. Section 5 and 6 briefly explain the implementation of our prototype tool and present our initial results. Finally, Section 7 covers future work and open research questions.

2 Dependence Graphs

We have proposed decorated dependence graphs as a more generic formalism for visualizing security properties, and performing static analysis of C

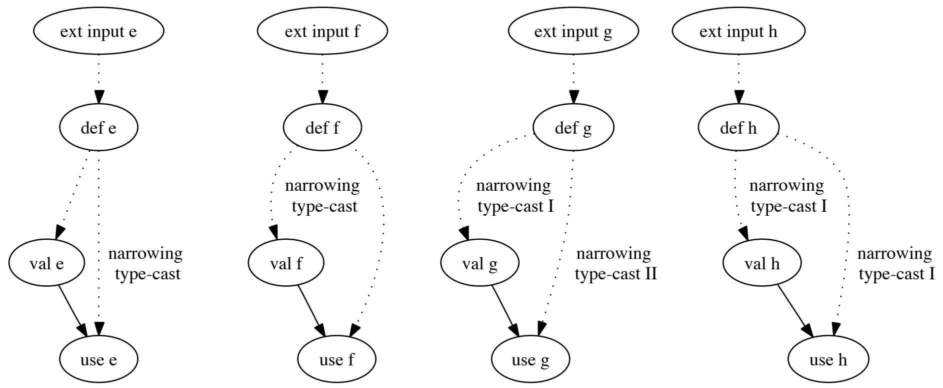


Figure 2: (Cont.) Last four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). "narrowing type-cast I" and "narrowing type-cast II" means two *different* type-casts. Severity ranking from left to right stretching from narrowing type-cast only before use and not in validation to validation with implicit narrowing type-casts.

code [154].

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [145]. Vertices represent statements and predicates (program points), and edges represent control- and data-flow *dependence*. A program point B is *control dependent* on another program point A, if A controls whether B is executed or not. A program point B is *data dependent* on a program point A if some variable x is defined in A and later used in B without any new defines in-between. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph. The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [147]. Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* [149].

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [134]. To detect such flaws we decorate the original SDGs be with type information, specifically implicit type conversions. Type conversion information belongs to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others.

Weber *et al* have used SDGs decorated with range constraint information for string buffers to statically detect buffer overflow vulnerabilities [139]. We will use this technique to check both buffer and integer ranges.

3 Integer Flaws

Several security vulnerabilities prove that handling integers is difficult. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and/or when the integer representation changes from signed to unsigned or vice versa [136, 151, 152]. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing type-casts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

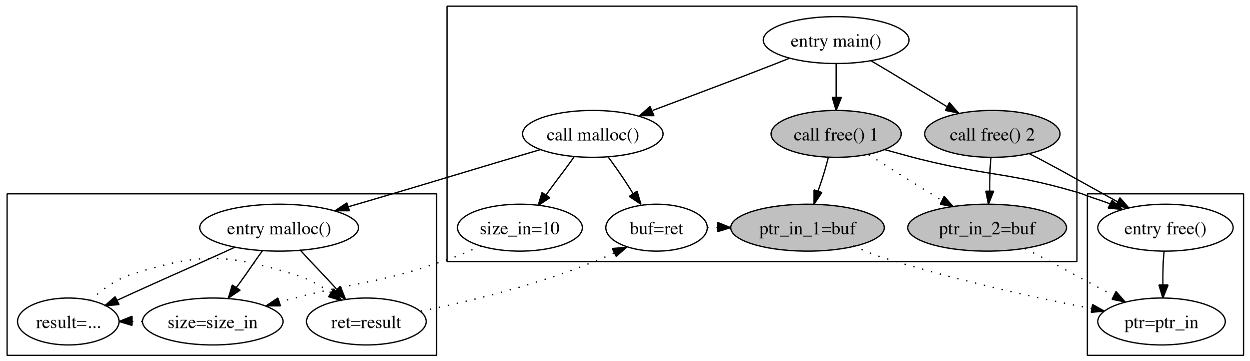


Figure 3: Incorrect graph pattern for malloc and free, where free is called twice. The grey nodes are the bad programming model where two free use the same pointer (shown by a data dependence).

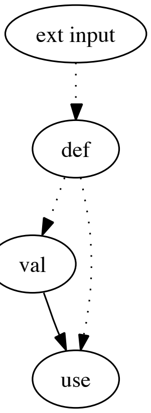


Figure 4:
Correct code pattern for integer input validation.

The graph to the left is an example of a model of good programming practice—correct integer input validation. The integer has to be validated before it is used which means that the use point (use) has to be control dependent on the validation point (val), and both use point and validation point have to be data dependent on the input without narrowing typecasts. Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [136].

To allow for severity ranking of reported flaws we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 1 and 2).

How to model external input is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. Apart from file accesses and command line arguments we have followed the pointers by Wheeler who mentions *Environment variables* as untrustworthy sources [153], and Ashcraft and Engler who add another three categories—System calls, routines that copy data from user space, and network data [136].

4 The Double `free()` Flaw

To allocate heap memory, a C program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`. To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored “in-band”. When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault*. But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of bad programming practice. We show in Fig. 3 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. If we ignore one of the calls to `free()` we have a match for correct usage of `free()`. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`, we also have to look for bad use of `free()`.

5 Tool Implementation

We have implemented a prototype tool called *GraphMatch* that performs pattern matching using dependence graphs. We build graph models of the programs with Grammatech’s tool *CodeSurfer* [12]. Currently GraphMatch can detect integer input validation flaws by following a straight forward algorithm (compare with vertex labels in Fig. 4):

1. Begin at some external input vertex (ext input)
2. Follow transitive data-flow to match definitions (def)
   (a) Follow data-flow to all sensitive uses (use)
   (b) Follow data-flow to all validations (val)
3. Check that all the sensitive uses from 2(a) are control-dependent on some validation in 2(b)

If some part of the program model deviates from the model of correct integer input validation it is reported as a potential flaw. This algorithm has a complexity of $O(E * V^h)$, where E and V are the number of edges and vertices in the program model, and h is the depth of the security property model.

6 Initial Results

The GraphMatch prototype performs well on our synthesized micro benchmarks whereas real-life applications pose a harder problem. We checked wu-ftpd 2.6-4 which consists of approx. 20.000 lines of code and produces a dependency graph with approx. 130.000 vertices. An analysis for integer input validation flaws took 15h on a 2.66 GHz Pentium 4. GraphMatch produced three warnings, two false positives and one true positive. The false positives were due to inaccuracy of our “sensitive use” model. The true positive was clearly a missing input validation but didn’t seem exploitable.

7 Future Work

Defining the modeling formalism was the first step toward a single tool able to check for several security properties. Apart from modeling other security properties and checking them with real-life code, we have several open research questions to address:

Complexity. Not too surprisingly, our initial results show that our graph matching has high complexity. It might be that dependence graph matching can be reduced to the *subgraph isomorphism problem* which is shown to be NP-complete [155]. Even so, we will investigate how heuristic trade-offs leading to unsoundness and/or incompleteness can affect practical performance.

Accuracy. How much does the inevitable inaccuracy of the underlying program analysis affect the accuracy of our pattern matching?

Generality. Are dependency graphs suitable for modeling a great variety of security properties of code? Are they suitable for analysis of other languages than procedural ones such as C?

Usability. Can visualization of code properties with dependence graphs help the programmers fix vulnerable code? Can it help in secure programming education?

Heuristic Ranking. Can we find effective heuristics for ranking of potential security bugs found through analysis?

Model Updates. Will our security property database be fairly static or will it need continuous updates with new flavors of the security properties?

Paper F

A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. Published in the Proceedings of the 10th Network & Distributed System Security Symposium 2003 (NDSS), 2003. The layout is modified.

A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention[1]

John Wilander and Mariam Kamkar
Dept. of Computer and Information Science, Linköpings universitet
{johwi, marka}@ida.liu.se

Abstract

The size and complexity of software systems is growing, increasing the number of bugs. Many of these bugs constitute security vulnerabilities. Most common of these bugs is the buffer overflow vulnerability. In this paper we implement a testbed of 20 different buffer overflow attacks, and use it to compare four publicly available tools for dynamic intrusion prevention aiming to stop buffer overflows. The tools are compared empirically and theoretically. The best tool is effective against only 50% of the attacks and there are six attack forms which none of the tools can handle.
Keywords: security intrusion; buffer overflow; intrusion prevention; dynamic analysis

1 Introduction

The size and complexity of software systems is growing, increasing the number of bugs. According to statistics from Coordination Center at Carnegie Mellon University, CERT, the number of reported vulnerabilities in software has increased with nearly 500% in two years [102] as shown in figure 1.

Now there is good news and bad news. The good news is that there is lots of information available on how these security vulnerabilities occur,

[1]This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

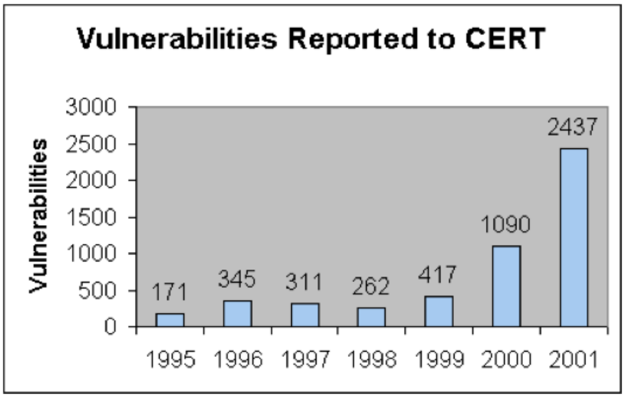


Figure 1: Software vulnerabilities reported to CERT 1995–2001.

how the attacks against them work, and most importantly how they can be avoided. The bad news is that this information apparently does not lead to fewer vulnerabilities. The same mistakes are made over and over again which, for instance, is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows stand for about 50% of the vulnerabilities reported by CERT [5].

In the middle of January 2002 the discussion about responsibility for security intrusions took a new turn. The US National Academies released a prepublication recommending that policy-makers create laws that would hold companies accountable for security breaches resulting from vulnerable products [103], which received global media attention [104, 105]. So far, only the intruder can be charged in court. In the future, software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

One starting point would, or could be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most likely be more effective, but given the amount of existing software ("legacy code"), the desire for modular design using software components programmed earlier, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools that aim to clean up vulnerable source code are necessary. A further discussion of this issue can be found in the January/February 2002 issue of IEEE Software [106].

In this paper we investigate the effectiveness of four publicly available tools for dynamic prevention of buffer overflow attacks–namely the GCC compiler patches StackGuard, Stack Shield, and ProPolice, and the security library Libsafe/Libverify. Our approach has been to first develop an in-depth understanding of how buffer overflow attacks work and from this knowledge build a testbed with all the identified attack forms. Then the four tools are compared theoretically and empirically with the testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [107].

1.1 Scope

We have tested publicly available tools for run-time prevention of buffer overflow attacks. The tools all apply to C source code, but using them requires no modifications of the source code. We do not consider approaches that use system specific features, modified kernels, or require the user to install separate run-time security components. The twenty buffer overflows represent a sample of the potential instances of buffer overflow attacks and not on the likelihood of a specific attack using the sample instance.

1.2 Paper Overview

The rest of the paper is organized as follows. Section 2 describes process memory management in UNIX and how buffer overflow attacks work. Section 3 presents the concept of intrusion prevention and describes the techniques used in the four analyzed tools. Section 4 defines our testbed of twenty attack forms and presents our theoretical and empirical comparison of the tools' effectiveness against the previously described attack forms. Section 5 describes the common shortcomings of current dynamic intrusion prevention. Finally sections 6 and 7 present related work and our conclusions.

2 Attack Methods

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according to definitions in, for example, the Internet Security Glossary [80]. In our context an intrusion or a successful attack aims to *change the flow of control*, letting the attacker execute arbitrary code. We consider this class of vulnerabilities the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation.

2.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function that writes to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes of space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing the flow of control occurs by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular target is the return address on the stack. But programmer defined *function pointers* and so called *longjmp buffers* are equally effective targets of attack.

2.2 Memory Layout in UNIX

To get a picture of the memory layout of processes in UNIX we can look at two simplified models (for a complete description see "Memory Layout in Program Execution" by Frederick Giasson [156]). Each process has a (partial) memory layout as in the figure below:

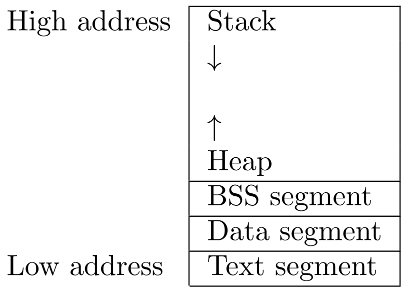


Figure 2: Memory layout of a UNIX process.

The machine code is stored in the text segment and constants, arguments, and variables defined by the programmer are stored in the other memory areas. A small C-program shows this (the comments show where each piece of data is stored in process memory):

```
static int GLOBAL_CONST = 1;      // Data segment
static int global_var;            // BSS segment

// argc & argv on stack, local
int main(argc **argv[]) {
  int local_dynamic_var;          // Stack
  static int local_static_var;    // BSS segment
  int *buf_ptr=(int *)malloc(32); // Heap
... }
```

For each function call a new *stack frame* is set up on top of the stack. It contains the return address, the calling function's base pointer, locally declared variables, and more. When the function ends, the return address instructs the processor where to continue executing code and the stored base pointer gives the offset for the stack frame to use.

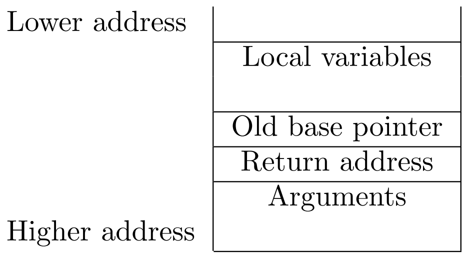


Figure 3: The UNIX stack frame.

2.3 Attack Targets

As stated above the target for a successful change of control flow is a code pointer. There are three types of code pointers to attack [157]. But Hiroaki Etoh and Kunikazu Yoda propose using the old base pointer as an attack target [158]. We have implemented their proposed attack form and proven that the old base pointer is just as dangerous a target as the return address (see section 2.4 and 4). So we have four attack targets:

1. The return address, allocated on the stack.
2. The old base pointer, allocated on the stack.
3. Function pointers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.
4. Longjmp buffers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.

A function pointer in C is declared as `int (*func_ptr) (char)`, in this example a pointer to a function taking a `char` as input and returns an `int`. It points to executable code.

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Let's say function A first calls `setjmp()`, then calls function B which in turn calls function C. If C now calls `longjmp()` the control is directly transferred back to function A, popping both C's and B's stack frames of the stack.

2.4 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [5, 108] and have been extensively analyzed and described in several papers and on-line documents [62, 109, 110, 111]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control either by directly overflowing the code pointer or by first overflowing another pointer and redirect that pointer to the code pointer.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

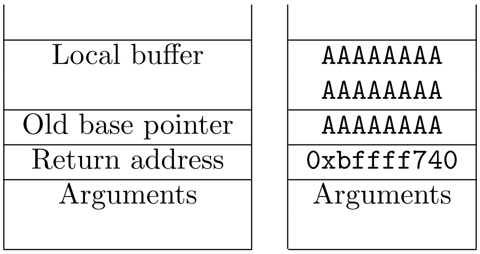


Figure 4: A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

The second attack target, the old base pointer, can be abused by building a fake stack frame with a return address pointing to attack code and then overflow the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.

The third attack target is function pointers. If the function pointer is redirected to the attack code the attack will be executed when the function pointer is used.

The fourth and last attack target is longjmp buffers. They contain the environment data required to resume execution from the point `setjmp()` was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code the attack will be executed when `longjmp()` is called.

Combining all these buffer overflow techniques, locations in memory and attack targets leaves us with no less than twenty attack forms. They are all listed in section 4 and constitute our testbed for testing of the intrusion prevention tools.

3 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [7] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention*. In this section we will first describe the differences between these two categories. Secondly, we describe four publicly available tools for dynamic intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against the intrusions and vulnerabilities described in section 2.4. This is not a complete survey of intrusion prevention techniques, rather a subset with the following constraints:

- Techniques used in the implementation phase of the software.
- Techniques that require no altering of source code to disarm security vulnerabilities.
- Techniques that are generic, implemented and publicly available, not prototypes or system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

3.1 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [28] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible targets. Static intrusion prevention removes the attacker's method of entry, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database of programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued.

3.2 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks.

Dynamic intrusion prevention, as we will see, often ends up becoming an intrusion detection system building on program and/or environment specific solutions, terminating execution in case of an attack. The techniques are often complete in the way that they can provably secure the targets they are designed to protect (one proof can be found in a paper by Chiueh and Hsu [159]) and will produce no false positives. Their general weakness lies in the fact that they all try to solve *known* security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. Whenever an attacker has figured out a new way of exploiting a bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs using the same attack method.

3.3 StackGuard

The *StackGuard* compiler invented and implemented by Crispin Cowan et al [160] is perhaps the most well referenced of the current dynamic intrusion prevention techniques. It is designed for detecting and stopping stack-based buffer overflows targeting the return address.

The StackGuard Concept

The key idea behind StackGuard is that buffer overflow attacks overwrite everything on their way towards their target. In the case of a buffer overflow on the stack targeting the return address, the attacker has to fill the buffer, then overwrite any other local variables below (i.e. on higher stack addresses), then overwrite the old base pointer until it finally reaches the return address. If we place a dummy value in between the return address and the stack data above, and then check whether this value has been overwritten or not before we allow the return address to be used, we could detect this kind of attack and possibly prevent it. The inventors have chosen to call this dummy value the *canary*.

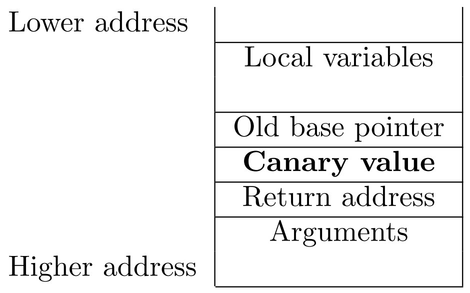


Figure 5: The StackGuard stack frame.

A potentially successful attack against such a system would be to somehow leave the canary intact while changing the return address, either by overwriting the canary with its correct value and thus not changing it, or by overwriting the return address through a pointer, not touching the canary. To solve the first problem, two canary versions have been suggested—firstly the *random canary* which consists of a random 32-bit value calculated at run-time, and secondly the *terminator canary* which consists of all four kinds of string termination sequences, namely `Null`, `Carriage Return`, `-1` and `Line Feed`. In the random canary case the attacker has to guess, or somehow retrieve, the random value at run-time. In the terminator canary case the attacker has to input all the termination sequences to keep the canary intact during the overflow. This is not possible since the string function receiving the input will terminate on one of the sequences.

Note that these techniques only stop overflow attacks that overwrite everything along the stack, not general attacks against the return address. The attacker can still abuse a pointer, making it point at the return address and writing a new address to that memory position. This shortcoming of StackGuard was discovered by Mariusz Woloszyn, alias "Emsi" and presented by Bulba and Kil3er [161]. The StackGuard team has addressed this problem by not only saving the canary value but the `XOR` of the canary and the correct return address. In this way an abused return address with an intact canary preceding it would still be detected since the `XOR` of the canary and the return address has changed. If the `XOR` scheme is used the canary has to be random since the terminator canary `XOR`ed with an address would not terminate strings anymore.

Random Canaries Unsupported

While testing StackGuard we noticed that the compiler did not respond to the flag set for random canary. We e-mailed Crispin Cowan and according to him: “There is only one threat that the XOR canary defeats, and the terminator canary does not: Emsi’s attack. However, if you have a vulnerability that enables you to deploy Emsi’s attack, then you have many other targets to attack besides function return address values. Therefore, we dropped support for random canaries [162]”. We agree that the return address is not the only attack target but it is the most popular and unlike function pointers and longjmp buffers, the return address is always present. According to Cowan’s e-mail and a WireX paper a better solution is on its way called *PointGuard* which will protect the integrity of pointers in general with the same kind of canary solution [157]. This implies that PointGuard will protect against all attack forms overflowing pointers (See attack forms 3a–f and 4a–f in section 4).

StackGuard is available for download at `http://www.immunix.org/`.

3.4 Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [163]. In the current version 0.7 it implements three types of protection, two against overwriting of the return address (both can be used at the same time) and one against overwriting of function pointers.

Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for Stack Shield. It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the return address on the normal stack is replaced by the copy on the Global Ret Stack. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the Global Ret Stack. This means only prevention and no detection of an attack. The Global Ret Stack has by default 256 entries which limits the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

Ret Range Check

A somewhat simpler but faster version of Stack Shield's protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted. Note that the Ret Range Check can detect an attack as opposed to the Global Ret Stack described above.

Protection of Function Pointers

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process' memory. That's where the programmer is likely to have implemented the functions to point at. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code injected into the process, since injection of data only can be done into the data segment, the BSS segment, the heap, or the stack.

Stack Shield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated. This protection will give false positives if the programmer has intended to use dynamically allocated function pointers.

Stack Shield is available for download at `http://www.angelfire.com/ - sk/stackshield/`.

3.5 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented the perhaps most sophisticated compiler protection called *ProPolice* [158].

The ProPolice Concept

Etoh's and Yoda's GCC patch ProPolice borrows the main idea from StackGuard (see section 3.3)—they use canary values to detect attacks on the stack. The novelty is the protection of stack allocated variables by rearranging the local variables so that `char` buffers always are allocated at the bottom, next to the old base pointer, where they cannot be overflown to harm any other local variables.

Building a Safe Stack Frame

After a program has been compiled with ProPolice the stack frame of functions look like that shown in figure 6.

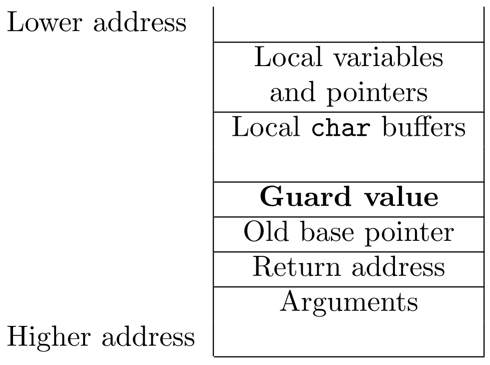


Figure 6: The ProPolice stack frame.

No matter in what order local variables, pointers, and buffers are declared by the programmer, they are rearranged in stack memory to reflect the structure shown above. In this way we know that local `char` buffers can only be overflown to harm each other, the old base pointer and below.

Function	Vulnerability
`strcpy(char *dest, const char *src)`	May overflow `dest`
`strcat(char *dest, const char *src)`	May overflow `dest`
`getwd(char *buf)`	May overflow `buf`
`gets(char *s)`	May overflow `s`
`[vf]scanf(const char *format, ...)`	May overflow arguments
`realpath(char *path, char resolved_path[])`	May overflow `path`
`[v]sprintf(char *str, const char *format, ...)`	May overflow `str`

Table 1: Vulnerable C functions that Libsafe adds protection to.

No variables can be attacked unless they are part of a `char` buffer. And by placing the canary which they call the *guard* between these buffers and the old base pointer all attacks outside the `char` buffer segment will be detected. When an attack is detected the process is terminated.

When testing ProPolice we noticed some irregularities in when and was not the buffer overflow protection was included. It seems like small char buffers (e.g. 5 bytes) confuse ProPolice, causing it to skip the protection even if the user has set the protector flag. This gives the overall impression maybe that ProPolice is somewhat unstable.

ProPolice is available for download at `http://www.trl.ibm.com/-projects/security/ssp/`.

3.6 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [117] is *Libsafe*. This tool actually provides a combination of static and dynamic intrusion prevention. Statically it patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten. Further protection has been provided [164] with *Libverify* using a similar dynamic approach to StackGuard (see Section 3.3).

Libsafe

The key idea behind Libsafe is to estimate a safe boundary for buffers on the stack at run-time and then check this boundary before any vulnerable function is allowed to write to the buffer. Vulnerable functions they consider to be the ones in table 1 below.

As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address.

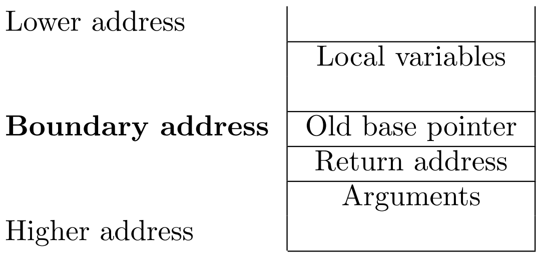


Figure 7: The Libsafe stack frame.

This boundary is enforced by overloading the functions in table 1 with wrapping functions. These wrappers first compute the length of the input as well as the allowed buffer size (i.e. from the buffer's starting point to the old base pointer) and then performs a boundary check. If the input is within the boundary the original functionality is carried out. If not the wrapper writes an alert to the system's log file and then halts the program. Observe that overflows within the local variables on the stack, such as function pointers, are not stopped.

Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackGuard. But since this is a library it does not require recompilation of the software. As with Libsafe the library is pre-loaded and linked to any program running on the system.

The key idea behind Libverify is to alter all functions in a process so that the first thing done in every function is to copy the return address onto a *canary stack* located on the heap, and the last thing done before returning is to verify the return address by comparing it with the address saved on the canary stack. If the return address is still correct the process is allowed to continue executing. But if the return address does not match the saved copy, execution is halted and a security alert is raised. Libverify does not protect the integrity of the canary stack. They propose protecting it with `mprotect()` as in RAD (see section 3.7) but as in the RAD case this will most probably impose a very serious performance penalty [159].

To be able to do this, Libverify has to rearrange the code quite a bit. First each function is copied whole to the heap (requires executable heap) where it can be altered. Then the saving and verifying of the return address is injected into each function by overwriting the first instruction with a call to `wrapper_entry` and all return instructions with a call to `wrapper_exit`. The need for copying the code to the heap is due to the Intel CPU architecture. On other platforms this could be solved without copying the code [164].

Libverify is needed to give a more complete protection of the return address since Libsafe only addresses standard C library functions (as pointed out by Istvan Simon [165]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

Libsafe and Libverify are available for download at `http://www.-research.avayalabs.com/project/libsafe/`.

3.7 Other Dynamic Solutions

The dynamic intrusion prevention techniques presented above are not the only ones. Other researchers have had similar ideas and implemented alternatives.

Tzi-cker Chiueh and Fu-Hau Hsu from State University of New York at Stony Brook have presented a compiler patch for protection of the return address [159]. They call their GCC patch *Return Address Defender*, or *RAD* for short. The key idea behind RAD is quite similar to the return address protection of Stack Shield described in Section 3.4. Every time a function call is made and a new stack frame is created, RAD stores a copy of the new return address. When a function returns, the return address about to be dereferenced is first checked against its copy. RAD is not publicly available.

The GCC patch *StackGhost* [166] by Mike Frantzen and Mike Shuey makes use of system specific features of the Sun Sparc Station to implement a sophisticated protection of the return address. They propose both XORing a random value with the return address (as StackGuard) as well as keeping a separate return address stack (as Stack Shield, RAD and Libverify). They also suggest using cryptographic methods instead of XOR to enhance security.

CCured and Cyclone are two recent research projects aiming to significantly enhance type and bounds checking in C. They both use a combination of static analysis and run-time checks.

CCured [167, 168] is an extension of the C programming language that distinguishes between various kinds of pointers depending on their usage. The purpose of this distinction is to be able to prevent improper usage of pointers and thus to guarantee that programs do not access memory areas they shouldn't access. CCured will change C programs slightly so that they are type safe. CCured does not change code that does not use pointers or arrays.

Cyclone [169] is a C dialect that prevents safety violations such as buffer overflows, dangling pointers, and format string attacks by ruling out certain parts of ANSI C and replacing them with safer versions. For instance `setjmp()` and `longjmp()` are unsupported (in some cases exceptions are used instead). Also pointer arithmetic is restricted. An average of 10% of the lines of code have to be changed when porting programs from C to Cyclone.

Richard Jones and Paul Kelly 1997 presented a GCC compiler patch in which they implemented run-time bounds checking of variables [170]. For each declared storage pointer they keep an entry in a table where the base and limit of the storage is kept. Before any pointer arithmetic or pointer dereferencing is made, the base and limit is checked in the table. While not explicitly aimed for security, this technique would effectively stop all kinds of buffer overflow attacks. Sadly their solution suffered both from performance penalties of more than 400 %, as well as incompatibilities with real-world programs (according to Crispin Cowan et al [171]). Because of the bad performance and compatibility we considered Jones' and Kelly's solution less interesting for software development and excluded it from our test.

It is also possible to have support for dynamic intrusion prevention in the operating system. A popular idea is the non-executable stack. This would make injection of attack code into the stack useless. But there are many ways around this protection. A few examples include using code already linked into the program from libraries (for instance calling `system()` with the parameter `"/bin/sh"`), injecting the attack code into other memory structures such as environment variables, or by exploiting buffer overflows on the heap or in the BSS/data segment. The Linux kernel patch from the Openwall Project is publicly available and implements a non-executable stack as well as protection against attacks using library functions [172]. Since it is a kernel patch it is up to the user and not the producer of software to install it. Therefore we did not include it in our test.

David Wagner and Drew Dean have presented an interesting approach for intrusion detection that relates to the functionality of the tools described in this paper [173]. They model the program's correct execution behavior via static analysis of the source code, building up callgraphs or even equivalent context-free languages defining the set of possible system call traces. Then these models are used for run-time monitoring of execution. Any deviation from the defined 'good' behavior will make the model enter an unaccepting state and trigger the intrusion alarm. As the metric for precision in intrusion detection they propose the branching factor of the model. A low branching factor means that the attacker has few choices of what to do next if he or she wants to evade detection.

4 Comparison of the Tools

Here we define our testbed of twenty buffer overflow attack forms and then present the outcome of our empirical and theoretical comparison of the tools from section 3.2.

We define an attack form as a combination of a technique, a location, and an attack target. As described in section 2.3 we have identified two

Development Tool	Attacks prevented	Attacks halted	Attacks missed	Abnormal behavior
StackGuard Terminator Canary	0 (0%)	3 (15%)	16 (80%)	1 (5%)
Stack Shield Global Ret Stack	5 (25%)	0 (0%)	14 (70%)	1 (5%)
Stack Shield Range Ret Check	0 (0%)	0 (0%)	17 (85%)	3 (15%)
Stack Shield Global & Range	6 (30%)	0 (0%)	14 (70%)	0 (0%)
ProPolice	8 (40%)	2 (10%)	9 (45%)	1 (5%)
Libsafe and Libverify	0 (0%)	4 (20%)	15 (75%)	1 (5%)

Table 2: Empirical test of dynamic intrusion prevention tools. 20 attack forms tested. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

techniques, two types of location and four attack targets:

Techniques. Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations. The types of location for the buffer overflow are the stack or the heap/BSS/data segment.

Attack Targets. We have four targets—the return address, the old base pointer, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Considering all practically possible combinations gives us the twenty attack forms listed below.

1. Buffer overflow on the stack all the way to the target:
   (a) Return address
   (b) Old base pointer
   (c) Function pointer as local variable

   (d) Function pointer as parameter
   (e) Longjmp buffer as local variable
   (f) Longjmp buffer as function parameter

2. Buffer overflow on the heap/BSS/data all the way to the target:
   (a) Function pointer
   (b) Longjmp buffer

3. Buffer overflow of a pointer on the stack and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter

4. Buffer overflow of a pointer on the heap/BSS/data and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter

Note that we do not consider differences in the likelihood of certain attack forms being possible, nor current statistics on which attack forms are most popular. However, we have observed that most of the dynamic intrusion prevention tools focus on the protection of the return address. Bulba and Kil3r did not present any real-life examples of their attack forms that

Development Tool	Attacks prevented	Attacks halted	Attacks missed
StackGuard Terminator Canary	0 (0%)	4 (20%)	16 (80%)
StackGuard Random XOR Canary	0 (0%)	6 (30%)	14 (70%)
Stack Shield Global Ret Stack	6 (30%)	7 (35%)	7 (35%)
Stack Shield Range Ret Check	0 (0%)	10 (50%)	10 (50%)
Stack Shield Global & Range	6 (30%)	7 (35%)	7 (35%)
ProPolice	8 (40%)	3 (15%)	9 (45%)
Libsafe and Libverify	0 (0%)	6 (30%)	14 (70%)

Table 3: Theoretical comparison of dynamic intrusion prevention tools. 20 attack forms used. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

defeated StackGuard and Stack Shield. Also the Immunix operating system (Linux hardened with StackGuard and more) came in second place at the Defcon “Capture the Flag” competition where nearly 100 crackers and security experts tried to compromise the competing systems [174]. This implies that the tools presented here are effective against many of the currently used attack forms. The question is: will this change as soon as this kind of protection is wide spread?

Also worth noting is that just because an attack form is prevented or halted does not mean that the very same buffer overflow can not be abused in another attack form. All of these attack forms have been implemented on the Linux platform and the source code is available from our homepage: `http://www.ida.liu.se/~johwi`.

To set up the test, the source code was compiled with StackGuard, Stack Shield, or ProPolice, or linked with Libsafe/Libverify. The overall results are shown in table 2. We also made a theoretical comparison to investigate the potential of the ideas and concepts used in the tools. The overall results of the theoretical analysis are shown in table 3. For details of the tests see appendix B and C.

Most interesting in the overall test results is that the most effective tool, namely ProPolice, is able to prevent only 50% of the attack forms. Buffer overflows on the heap/BSS/data targeting function pointers or longjmp buffers are not prevented or halted by any of the tools, which means that a combination of all techniques built into one tool would still miss 30% of the attack forms.

This however does not comply with the result from the theoretical comparison. Stack Shield was not able to protect function pointers as stated by Vendicator. Another difference is the abnormal behavior of StackGuard and Stack Shield when confronted with a fake stack frame in the BSS segment.

These poor results are all evidence of the weakness in dynamic intrusion prevention discussed in section 3.2, the tested tools all aim to protect *known* attack targets. The return address has been a popular target and therefore all tools are fairly effective in protecting it.

Worth noting is that StackGuard halts attacks against the old base pointer although that was not mentioned as an explicit design goal.

Only ProPolice and Stack Shield offer real intrusion prevention—the other tools are more or less intrusion detection systems. But still the general behavior of all these tools is termination of process execution during attack.

5 Common Shortcomings

There are several shortcomings worth discussing. We have identified four generic problems worth highlighting, especially when considering future research in this area.

5.1 Denial of Service Attacks

Since three out of four tools terminate execution upon detecting an attack they actually offer more of intrusion detection than intrusion prevention. More important is that the vulnerabilities still allow for Denial of Service attacks. Terminating a web service process is a common goal in security attacks. Process termination results in a much less serious attack but will still be a security issue.

5.2 Storage Protection

Canaries or separate return address stacks have to be protected from attacks. If the canary template or the stored copy of the return address can be tampered with, the protection is fooled. Only StackGuard with the terminator canary offers protection in this sense. The other tools have no protection implemented and the performance penalty of such protection can be very serious—up to 200 times [159].

5.3 Recompilation of Code

The three compiler patches have the common shortcoming of demanding recompilation of all code to provide protection. For software vendors shipping new products this is a natural thing but for running operating systems and legacy systems this is a serious drawback. Libsafe/Libverify offers a much more convenient solution in this sense. The StackGuard and ProPolice teams have addressed this issue by offering protected versions of Linux and FreeBSD.

5.4 Limited Nesting Depth

When keeping a separate stack with copies of return addresses, the nesting depth of the process is limited. Only Vendicator, author of Stack Shield, discusses this issue but offers no real solution to the problem.

6 Related Work

Three other studies of defenses against buffer overflow attacks have been made.

In late 2000 Crispin Cowan et al published their paper "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" [157]. They implicitly discuss several of our attack forms but leave out the old base pointer as an attack target. Comparison of defenses is broader considering also operating system patches, choice of programming language and code auditing but there is only a theoretical analysis, no comparative testing is done. Also the only dynamic tools discussed are their own StackGuard and their forthcoming PointGuard.

Only a month later Istvan Simon published his paper “A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks” [165]. It discusses pros and cons with operating system patches, StackGuard, Libsafe, and similar solutions. The major drawback in his analysis is the lack of categorization of buffer overflow attack forms (only three of our attack forms are explicitly mentioned) and any structured comparison of the tool’s effectiveness. No testing is done.

In March 2002 Pierre-Alain Fayolle and Vincent Glaume published their lengthy report “A Buffer Overflow Study, Attacks & Defenses” [175]. They describe and compare Libsafe with a non-executable stack and an intrusion detection system. Tests are performed for two of our twenty attack forms. No proper categorization of buffer overflow attack forms is made or used for testing.

7 Conclusions

There are several run-time techniques for stopping the most common of security intrusion attack—the buffer overflow. But we have shown that none of these can handle the diverse forms of attacks known today. In practice at best 40% of the attack forms were prevented and another 10% detected and halted, leaving 50% of the attacks still at large. Combining all the techniques in theory would still leave us with nearly a third of the attack forms missed. In our opinion this is due to the general weakness of the dynamic intrusion prevention solution—the tools all aim at protecting *known* attack targets, not all targets. Nevertheless these tools and the ideas they are built on are effective against many security attacks that harm software users today.

8 Acknowledgments

We are grateful to the readers who have previewed and improved our paper, especially Crispin Cowan.

Paper G

RIPE: Runtime Intrusion Prevention Evaluator. Published in the Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011), December 5-9, 2011. The layout is modified.

RIPE: Runtime Intrusion Prevention Evaluator

John Wilander
Dept. of Computer Science
Linköpings universitet
john.wilander@gmail.com

Nick Nikiforakis
Katholieke Universiteit Leuven
Belgium
nick.nikiforakis@cs.kuleuven.be

Yves Younan
K. U. Leuven
Belgium
yves.younan@cs.kuleuven.be

Mariam Kamkar
Dept. of Computer Science
Linköpings universitet
mariam.kamkar@liu.se

Wouter Joosen
K. U. Leuven
Belgium
wouter.joosen@cs.kuleuven.be

Abstract

Despite the plethora of research done in code injection countermeasures, buffer overflows still plague modern software. In 2003, Wilander and Kamkar published a comparative evaluation on runtime buffer overflow prevention technologies using a testbed of 20 attack forms and demonstrated that the best prevention tool missed 50% of the attack forms. Since then, many new prevention tools have been presented using that testbed to show that they performed better, not missing any of the attack forms. At the same time though, there have been major developments in the ways of buffer overflow exploitation.

In this paper we present *RIPE*, an extension of Wilander's and Kamkar's testbed which covers 850 attack forms. The main purpose of RIPE is to provide a standard way of testing the coverage of a defense mechanism against buffer overflows. In order to test RIPE we use it to empirically evaluate some of the newer prevention techniques. Our results show that the most popular, publicly available countermeasures cannot prevent all of RIPE's buffer overflow attack forms. ProPolice misses 60%, Libsafe-Plus+TIED misses 23%, CRED misses 21%, and Ubuntu 9.10 with non-executable memory and stack protection misses 11%.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*;

D.2.8 [**Software Engineering**]: Metrics—*Product metric*

1 Introduction

Buffer overflows are probably the single most well-known exploitation technique in the history of computer security. The ability to take control of the execution flow of a process by overwriting adjacent data, first received world-wide attention through the Morris worm [176] and since then has been used as the exploitation mechanism in most of the well-known worms (e.g. CodeRed [177] and SQLSlammer [178]) as well as in countless attacks against popular software. Due to the severity and popularity of this attack, researchers have produced a significant amount of papers describing techniques which can, among others, detect, defend, stop and heal buffer-overflows at all possible stages of a program's lifetime. A small number of these suggested techniques have reached production level, by being included in popular programming frameworks and operating systems.

Despite the amount of research conducted in the area of buffer overflows in specific, and code injection techniques in general, modern software is still plagued by buffer-overflows which are discovered, almost weekly, in one of the many popular software products. At the time of this writing, the US-CERT [179] reports that 14 buffer-overflow vulnerabilities have been found in 2011, many of which are in products of Microsoft, Adobe and Google. This shows that the problem of buffer overflows is far from resolved and that researchers in both academia and industry should focus their efforts in creating countermeasures that can eventually be part of real running systems.

While there are standard ways to measure the performance overhead of a buffer overflow countermeasure, such as the SPEC CPU [180] and Olden benchmarks, there is no standard way of testing and comparing the defense coverage of any given countermeasure.

In 2003, Wilander and Kamkar [181] published a comparative evaluation on runtime buffer overflow prevention using a testbed of 20 attack forms and demonstrated that the best prevention tool missed 50% of the attack forms. That testbed has been used to demonstrate the effectiveness of subsequent tools and techniques [14, 15, 16, 17, 18, 19] and the outcome of the 2003 evaluation was used to motivate further preventive research [20, 21, 22, 23].

We believe that many of the attack techniques tested by that testbed are now outdated and thus a perfect “score” of a protection countermeasure against them is of limited value.

In this paper we introduce RIPE—*Runtime Intrusion Prevention Evaluator*—which comprises of 850 buffer overflow attack forms. The main purpose of RIPE is to quantify the protection coverage of any given countermeasure by performing a wide range of buffer-overflow attacks and recording their success or failure. The tool is released as free software (see Section 10 for availability) in an attempt to standardize the comparison between countermeasures and to further support research on code-injection countermeasures. In order to test the applicability and usability of the RIPE testbed, we tested it against buffer overflow countermeasures that the authors have made publicly available or that were kindly provided to us.

This paper and the release of the RIPE testbed provides the following research contributions:

1. Implementation of the combinatorial set of buffer overflow attack forms built on 4 locations of buffers in memory, 16 target code pointers, 2 overflow techniques, 5 variants of attack code being executed, and 10 functions being abused. In total, 850 working attack forms.

2. Empirical evaluation of publicly available buffer overflow countermeasures using canaries, boundary checking, copying and checking target data, library wrapping, and non-executable memory.

3. Open source, fully documented testbed code and driver engine for evaluation and reporting.

The rest of this paper is structured as follows: The RIPE testbed is described in Sections 2 and 3. Section 4 gives an overview of buffer overflow prevention techniques. Our evaluation setup is explained in Section 5 and the results are presented in Section 6. Section 7 contains related work. Finally, Section 8 describes future work and we conclude in Section 9.

2 The RIPE Buffer Overflow Testbed

The RIPE (Runtime Intrusion Prevention Evaluator) testbed has a backend built in C and a frontend built in Python. The backend or "attack generator" of RIPE lets the user dynamically specify which type of buffer overflow she wants to test. For instance:

```
./ripe_attack_generator -f strcpy -t direct -l stack
-c ret -i nonop
```

... will perform the standard stack smashing attack abusing the `strcpy()` function to overflow a buffer on the stack all the way to the return pointer, redirecting it to injected attack code without a NOP sled. As another example:

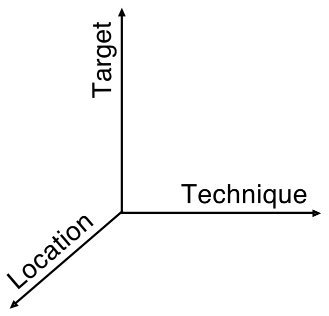
```
./ripe_attack_generator -f sscanf -t indirect -l heap
-c funcptrstackparam -i returnintolibc
```

... will abuse the `sscanf()` function to perform an overflow of a buffer located on the heap, overwrite a general pointer and make it point to a function pointer parameter (indirect attack), and redirect that function pointer to return-into-libc attack code. In the next sections, we will present RIPE's *dimensions*, which are essentially all the user-configurable parameters of an attack. Then we'll enumerate all the parameters that build up the combined 850 attack forms.

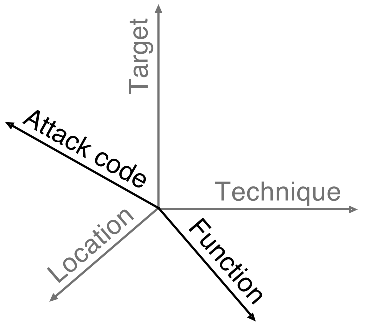
2.1 Testbed Dimensions

Wilander and Kamkar's testbed (hereafter refered to as the "NDSS'03 testbed") had three dimensions all of which are included in the new RIPE testbed too; *location* of buffer in memory, *target code pointer*, and *overflow technique* - Fig 4.1(a).

The new RIPE testbed has five dimensions including extended versions of the original three. The additions are *attack code* and *function abused* - Fig 4.1(b).



(a) Dimensions of NDSS'03 testbed



(b) Dimensions of RIPE testbed

Figure 1: The difference of dimensions supported by the NDSS'03 testbed and RIPE

2.2 Dimension 1: Location

The first testbed dimension is the memory location of the buffer to be overflowed. Both the NDSS'03 testbed and RIPE support four buffer locations; Stack, Heap, BSS, and Data segment.

2.3 Dimension 2: Target Code Pointer

The second testbed dimension is the target code pointer, i.e. the code pointer to redirect towards the attack code. RIPE supports the following target code pointers:

- *Return address*: The address stored by a function in order to return to the appropriate offset of the caller
- *Old base pointer*: The previous contents of the `EBP` register, which is used to reference function arguments and local variables
- *Function pointers*: Generic function pointers allowing programmers to dynamically call different functions from the same code
- *Longjmp buffers*: Setjmp/longjmp is a technique which allows programmers to easily jump back to a predefined point in their code (see Sec. 3.2).

- *Vulnerable Structs*: Structs which group a buffer and a function pointer and can be abused by attackers to overflow from one to the other (see Sec. 3.3).

With the exception of the *Return Address* and the *Old base pointer targets* that are Stack-specific, all other targets are allocated on all available data segments of a process, i.e. on the Stack (both as local variables and function arguments), on the Heap and on the Data/BSS segment.

2.4 Dimension 3: Overflow Technique

The third testbed dimension is overflow technique. Both the NDSS'03 testbed and RIPE support *Direct* and *Indirect* overflowing techniques. In direct techniques, the target is adjacent to the overflowed buffer or can be reached by sequentially overflowing from the buffer. On the other hand, indirect overflowing makes use of generic pointers in a two-step approach. First the generic pointer is overflowed with the address of the target and then at a later dereference, the target is overwritten with attacker-controlled data. This technique was originally introduced by Bulba and Kil3r [161] as a way to bypass the StackGuard countermeasure. A pointer before the StackGuard canary was used to overwrite the return-address while maintaining the integrity of the canary.

2.5 Dimension 4: Attack Code

The fourth testbed dimension is new for RIPE – attack code. A user running the testbed can choose between attack code that spawns a shell on the vulnerable machine or attack code that creates a file in a specific directory. The former can be used when trying out individual attacks and the latter is used by the front-end part of RIPE which exhaustively tries all available attack combinations and then reports the full results. The variations of these two shellcodes are presented in the following list:

- *Shellcode without NOP sled*: This option can be useful in testing the accuracy of attacks as well as challenge countermeasures that rely on the detection of specific code patterns (such as the presence of a set of `0x90` bytes (NOP)) in the process' address space.

- *Shellcode with NOP sled*: This is the most-used form of shellcode that prepends the attacker's functionality with a set of NO-Operation instructions to improve the attacker's chances of correctly redirecting the execution-flow of the program into his injected code.
- *Shellcode with polymorphic NOP sled*: In this case, the NOP sled is not the standard set of `0x90` bytes but a set of instructions that can be executed without affecting the correctness of the actual attack code. As with the first variation, countermeasures that over-rely in the presence of standardized NOP-sleds will have difficulty countering such attacks. Akritidis et al. [182] conducted a study where, among others, they showed how obfuscation and encryption can be used by attackers to evade Network Intrusion Detection Systems (NIDS).
- *Return-into-libc*: Return-into-libc are attacks where the attacker does not inject new code in the process' address space but rather uses existing functions to perform his attack, for instance using the `system` libc-function to execute an interactive shell. This attack was essentially a natural evolution for attackers, when countermeasures that disallowed execution from writable memory pages, e.g. Data Execution Prevention (DEP) and $W \oplus$X, became popular in modern operating systems. RIPE uses `system()` for the spawning of an interactive shell and `creat()` for creating new files.
- *Return-Oriented Programming (ROP)*: Return-oriented programming [63] is the most recent way of carrying-out exploitation, once an attacker has achieved control of the execution flow. ROP is a generalization of Return-into-libc where now an attacker can use chunks of functionality from existing code (gadgets) and combine them to create new functionality. While we have implemented a ROP attack in our testbed, we haven't yet implemented stack-pivoting techniques and thus we can only trigger such an attack when we control the contents of the existing stack, as is the case in a stack-smashing attack.

2.6 Dimension 5: Function Abused

The fifth and final testbed dimension is also new for RIPE – function abused. A user can choose to perform the buffer overflow with `memcpy()`, `strcpy()`, `strncpy()`, `sprintf()`, `snprintf()`, `strcat()`, `strncat()`, `sscanf()`, `fscanf()`, and also with “homebrew”, a loop-based equivalent of `memcpy`.

The `n`-containing functions are designed to take the target buffer size into account which should prevent buffer overflows. The size however, is provided by the developer (static or calculated) and thus a miscalculation can cancel-out the protection offered by these functions. Known caveats include the fact that parameter `n` means *total* buffer size for `strncpy()` but *remaining* buffer space for `strncat()` [183], and if `n` is undefined for instance because of a `NULL` value in the length calculation `strncpy()` will allow for buffer overflow as shown in CVE-2009-4035 [184]:

```
line1 = getNext(line);  // May return NULL
if ((n = line1 - line) > 255) {
  n = 255;
}
strncpy(buf, line, n);  // n undef or < 0
```

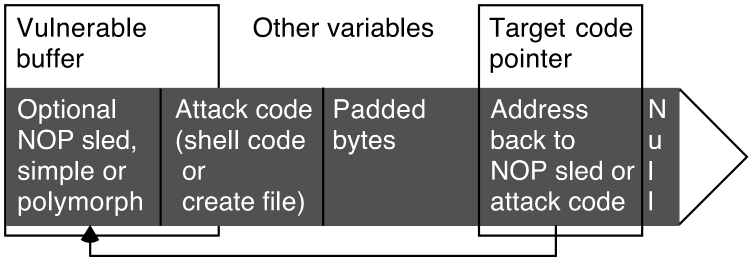
3 Building Payloads

RIPE’s attack generator dynamically builds the specified payload and performs the attack on itself, i.e., the code contains all the required vulnerable buffers and pointers as well as the logic for offsets, attack code and overflows.

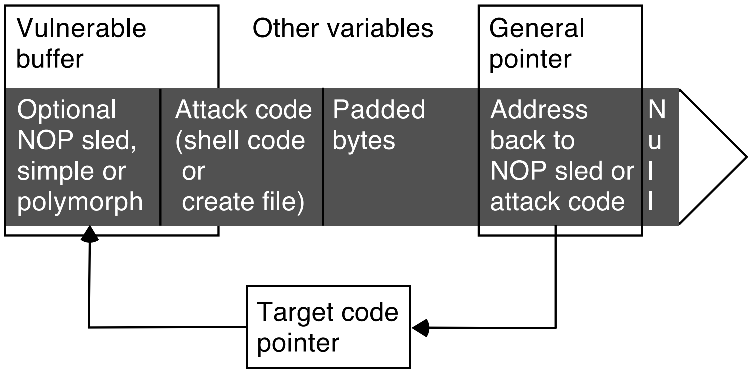
Figure 4.2(a) shows the payload of a direct overflow with injected code, and Figure 4.2(b) shows an *indirect* overflow using an intermediate pointer to target the code pointer.

3.1 Fake Stack Frame

The old base pointer is pushed on the stack immediately above the return address and is a possible target code pointer. The overflow redirects the base pointer towards an injected fake stack frame with a fake return pointer pointing to the attack code - Fig. 4.3(a).


Vulnerable buffer
Other variables
Target code pointer
Optional NOP sled, simple or polymorph
Attack code (shell code or create file)
Padded bytes
Address back to NOP sled or attack code
Null

(a) Direct Attack

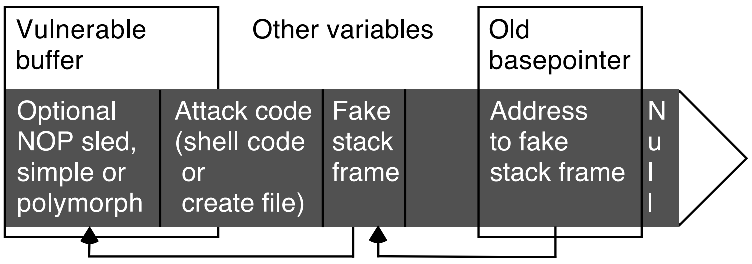

Vulnerable buffer
Other variables
General pointer
Optional NOP sled, simple or polymorph
Attack code (shell code or create file)
Padded bytes
Address back to NOP sled or attack code
Null
Target code pointer

(b) Indirect Attack

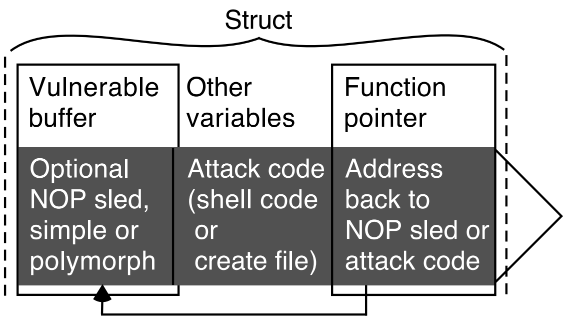
Figure 2: A direct and an indirect overflow payload with injected attack code

3.2 Longjmp Buffer

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Consider a program where function A first calls `setjmp()`, then calls function B which in turn calls function C. If C now calls `longjmp()` the control is directly transferred back to function A, popping both C's and B's stack frames of the stack. Longjmp buffers contain the environment data required to resume execution from the point `setjmp()` was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code


Vulnerable buffer
Other variables
Old basepointer
Optional NOP sled, simple or polymorph
Attack code (shell code or create file)
Fake stack frame
Address to fake stack frame
N u l l

(a) Fake Stack Frame


Struct
Vulnerable buffer
Other variables
Function pointer
Optional NOP sled, simple or polymorph
Attack code (shell code or create file)
Address back to NOP sled or attack code

(b) Overflowed struct

Figure 3: An attack where a fake stack frame (with an attacker-controlled return address) is created and an overflow of a function pointer from within the same struct

the attack will be executed when `longjmp()` is called.

3.3 Struct With Function Pointer

A struct containing a buffer and a function pointer can allow for an internal buffer overflow attack within the struct since there is no reordering of variables to make the code pointer unreachable from the buffer, nor are there any canary values between the buffer and the target code pointer - Fig 4.3(b). Such structs have been previously discussed by Zhivich et al [185].

4 Runtime Buffer Overflow Prevention

The research in countering buffer overflow attacks at runtime has gone in several directions. We have identified six general categories or techniques, namely canary-based, boundary checking, copying and checking target data, encrypted instruction addresses, library wrappers, and non-executable and randomized memory. Our evaluation covers all but encrypted instruction addresses since we were not able to locate a publicly-available countermeasure performing such operations.

4.1 Canary-Based Tools

This technique was invented by Cowan *et al* [160] and prevents buffer overflows by adding a *canary value* to sensitive memory regions. The canary's integrity is checked before the sensitive memory is used. If the canary has been changed the sensitive memory may have been corrupted and the program is typically terminated. Other tools have adopted the canary, for instance detection of heap-based overflows targeting malloc linked lists by Robertson *et al* [22], Microsoft's `/GC` compiler flag [186], and stack protection with ProPolice by Etoh *et al* [158]. ProPolice is covered in our empirical evaluation and is presented in more detail in Sec. 5.1.

4.2 Boundary Checking Tools

Standard C and C++ do not have runtime bounds checking unlike newer languages such as Java and C#. This is one of the fundamental design decisions that make buffer overflow attacks possible. Researchers have implemented variants of C compilers that include boundary checking in binaries.

In 1997 Jones and Kelly presented a GCC compiler patch in which they implemented runtime bounds checking of variables [170]. Sadly their solution suffered from performance penalties of more than 400%, as well as incompatibility with real-world programs [171]. Ruwase and Lam continued Jones' and Kelly's work and have implemented a GCC patch called "CRED" [18]. CRED is covered in our empirical evaluation and is presented in section 5.5.

4.3 Tools Copying and Checking Target Data

StackShield [163] and Libverify [164] were the first buffer overflow prevention tools that used the technique of storing copies of return addresses on a separate stack. When a function returned, its stored return address is checked against the copy on the separate stack. If the addresses differed either the correct address was copied back or execution was halted. StackShield is a compiler patch whereas Libverify patched the code during load. Both StackShield and Libverify are covered in our empirical evaluation and are presented in section 5.2 and 5.3.

Chiueh and Hsu [159] presented a compiler patch called *RAD* in 2001. It used a separate stack to keep copies of return addresses similar to StackShield. Smirnov and Chiueh have continued the work and implemented a more complex GCC patch called *DIRA* [187]. Apart from the separate stack with copies of return addresses, DIRA keeps copies of function pointer values in a special buffer.Nebenzahl and Wool [188] have developed a technique for instrumenting Windows binaries at install-time with a separate stack for copies of return addresses.

4.4 Library Wrappers

Buffer overflow prevention through library wrappers was originally done by Baratloo, Singh, and Tsai, and their tool was called *Libsafe* [117]. It patches library functions in C that constitute potential buffer overflow vulnerabilities. In the patched functions a range check is made before the actual function call. As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address.

Avijit, Gupta and Gupta continued the work by Baratloo *et al* by implementing *LibsafePlus* and *TIED* [189, 190]. Their system collects and stores information about the sizes of both stack and heap buffers. This information is then used at runtime to ensure that no character buffers are written past their limit. Libsafe, LibsafePlus, and TIED are all covered in our empirical evaluation and are presented in more detail in Sections 5.3, 5.4.

4.5 Non-Executable and Randomized Memory

The Linux kernel patch from the Openwall Project was the first to implement a non-executable stack [172]. Not allowing execution of code stored on the stack effectively stops execution of attack code injected on the stack and on the heap. In some cases, researchers have been able to circumvent this countermeasure by abusing certain traits of a program, e.g. convincing the Just-in-time compiler of ActionScript in Macromedia Flash to place attacker-code in their writable and executable memory pages [191].

Two more recent kernel patches that deny execution both on the stack and on the heap are PaX [192] and ExecShield [193]. They also randomize address offsets from the base of memory locations, called *Address Space Layout Randomization*, to further countermeasure buffer overflow attacks. DieHard [194] and its continuation DieHarder [195] are memory allocators which randomize the location of heap objects on the heap and require larger-than-needed address spaces to ensure probabilistic safety. Even though DieHard is publicly available we could not include it in our evaluation since RIPE is a process that attacks itself calculating the needed offsets from within its source code (see Sec. 8). This means, that RIPE would "unfairly" de-randomize DieHard and successfully perform all of the attacks.

5 Empirical Evaluation Setup

We have used RIPE to evaluate a number of preventive tools and techniques designed to counter buffer overflow attacks, namely ProPolice (canary-based), CRED (boundary checking), StackShield and Libverify (copying and checking target data), Libsafe, LibsafePlus, LibsafePlus+TIED (library wrappers), and PAE and XD (non-executable memory).

The theoretical number of attack forms produced by multiplying all the choices is 3,840 (4 locations * 16 target code pointers * 2 techniques * 3 variants of attack code without NOP sled variations * 10 functions being abused). However, that number incorporates 2,990 practically impossible attack forms. For instance it's not possible to perform a direct buffer overflow all the way from the BSS segment to the stack since the stack is not writable for a BSS segment variable. Thus, the number of working attack forms is 850. In the empirical evaluation we have left out the three NOP versions and only executed with one (the simple NOP sled). Since none of the protection tools or techniques evaluated tries to detect NOP sleds *per se* including them would not change the results of the current analysis. Nevertheless RIPE supports three different NOP sled settings for attack code.

5.1 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented a compiler protection called *ProPolice* [158]. It borrows the main idea from StackGuard—canary, or *guard* values to detect attacks on the stack. The guard is placed between the buffers and the old base pointer meaning it protects both the return pointer and the old base-pointer from direct overflows. In addition to the guard, ProPolice rearranges the local stack variables so that `char` buffers always are allocated at the bottom, next to the canary, where they cannot harm any other local variables if overflowed. Non-char buffer variables can only be attacked if they are part of a struct that also contains a buffer.

5.2 StackShield

StackShield is a compiler patch for GCC made by Vendicator [163]. In the current version 0.7 it implements three types of protection, two against overwriting of the return address (both can be used at the same time) and one against overwriting of function pointers.

Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for StackShield. It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the return address on the normal stack is replaced by the copy on the Global Ret Stack. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the Global Ret Stack allowing the countermeasure to prevent but not to detect buffer overflows (and possible corruption of data due to them). The Global Ret Stack has by default 256 entries which limits the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

Ret Range Check

A somewhat simpler but faster version of StackShield's protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted. Note that the Ret Range Check can detect an attack as opposed to the Global Ret Stack described above.

Protection of Function Pointers

StackShield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process' memory where the programmer is likely to have implemented the functions to point at. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code injected into the process, since injection of data only can be done into the stack, the heap, the BSS, or the data segment.

StackShield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated. This protection will give false positives if the program uses dynamically allocated function pointers.

5.3 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [117] is *Libsafe*. This tool actually provides a combination of static and dynamic intrusion prevention. Statically it patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten. Further protection has been provided [164] with *Libverify* using a similar dynamic approach to StackGuard.

Libsafe

The key idea behind Libsafe is to estimate a safe boundary for buffers on the stack at run-time and then check this boundary before any vulnerable function is allowed to write to the buffer.

As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address. This boundary is enforced by overloading `strcpy()`, `strcat()`, `getwd()`, `gets()`, `[vf]scanf()`, `realpath()`, and `[v]sprintf()` with wrapping functions. These wrappers first compute the length of the input as well as the allowed buffer size (i.e. from the buffer's starting point to the old base pointer) and then performs a boundary check. If the input is within the boundary the original functionality is carried out. If not the wrapper writes an alert to the system's log file and then halts the program. Observe that overflows within the local variables on the stack, such as function pointers, are not stopped.

Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackShield. However, since this is a library it does not require recompilation of the software. As with Libsafe the library is pre-loaded and linked to any program running on the system. The key idea behind Libverify is to alter all functions in a process so that the first thing done in every function is to copy the return address onto a *canary stack* located on the heap, and the last thing done before returning is to verify the return address by comparing it with the address saved on the canary stack. If the return address is still correct the process is allowed to continue executing. However, if the return address does not match the saved copy, execution is halted and a security alert is raised. Libverify does not protect the integrity of the canary stack. They propose protecting it with `mprotect()` like *Return Address Defender*, RAD [159]. However, as in the RAD case this will most probably impose a serious performance penalty.

To be able to do this, Libverify has to transform the code of a given program. First each function is copied whole to the heap (requires executable heap) where it can be altered. Then the saving and verifying of the return address is injected into each function by overwriting the first instruction with a call to `wrapper_entry` and all return instructions with a call to `wrapper_exit`. The need for copying the code to the heap is due to the Intel CPU architecture. On other platforms this could be solved without copying the code [164]. Libverify is needed to give a more complete protection of the return address since Libsafe only addresses standard C library functions (as pointed out by Istvan Simon [165]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

5.4 LibsafePlus and TIED

Avijit, Gupta and Gupta continued the work by Baratloo *et al* by implementing *LibsafePlus* and *TIED* [189, 190]. TIED collects static information and LibsafePlus collects dynamic information about the sizes of stack and heap buffers. This information is used runtime to ensure that no character buffers are written past their limit.

Static buffer sizes are collected compile-time by exploiting debugging information produced by a specific compiler option. Dynamic buffer size information is collected runtime by interception of calls to `malloc()` and `free()`. Finally, the original technique with wrappers for dynamically linked libraries handling strings is used to check the bounds. Their main contributions are a more precise boundary check of stack buffers than the previous solution, and a boundary check of heap buffers.

5.5 CRED

Ruwase and Lam continued Jones' and Kelly's boundary checking work and have implemented a GCC patch called "CRED", *C Range Error Detector* [18]. Their goals were for the runtime checks to impose less overhead and provide better compatibility. To enhance performance they only perform boundary checks on string buffers since they consider such buffers the most likely ones vulnerable to security attacks. With such a restriction most of the programs they tested suffered less than 26 % overhead. Worst case was a string-intensive email program which suffered 130 % overhead.

Compatibility was solved by storing out-of-bounds pointer values in so called *out-of-bounds objects.* If pointer arithmetics using the out-of-bounds pointer results in an in-bounds address the pointer is sanitized. All variables with a memory range such as arrays and structs get an associated *referent object* that keeps of pointer arithmetic and bounds. Pointer operations that reference memory outside the referent object are illegal. CRED allows out-of-bounds references to be part of arithmetic as long as the resulting access is within bounds.

5.6 Non-Executable Memory and Stack Protector (Ubuntu 9.10)

We have evaluated the buffer overflow prevention techniques used in Ubuntu 9.10 "Karmic" [196] which has several security features [197] relevant to buffer overflow prevention.

ASLR

Ubuntu 9.10 has five ASLR (Address Space Layout Randomization) features, four enabled by default—Stack ASLR, Libs/mmap ASLR, Exec ASLR, brk ASLR, and VDSO ASLR [197].

The fundamentals of defeating ASLR have been studied by Schacham *et al* [198]. Attackers may reduce the entropy present in a randomized address space by leaking information via format string attacks [199], buffer overreads [200] or covering multiple bits of entropy per attack by using heap spraying, introduced by Hassell and Permeh [201]. RIPE does not use brute force, information leakage, or heap spraying to circumvent ASLR. While such attack methods are interesting, RIPE currently focusses on evaluating countermeasures performing attack detection or active prevention, not on countermeasures making attacks harder. RIPE calculates its offsets and target addresses at runtime and thus has information available *after* randomization. The effects of this decision are discussed in detail in Section 8.

Non-Executable Memory

Most modern CPUs protect against executing non-executable memory regions (heap, stack, etc), known either as Non-eXecute (NX) or eXecute-Disable (XD) [202]. This protection reduces the areas an attacker can use to perform arbitrary code execution. It requires that the kernel uses PAE, *Physical Address Extension*. Ubuntu 9.10, partially emulates this protection for processors lacking NX when running on a 32bit kernel. Our evaluation runs where performed on a machine with an Intel Core 2 Duo processor with XD support enabled.

Stack Protector (ProPolice)

Ubuntu 9.10 ships with a patched gcc using `-fstack-protector` by default. This protection is ProPolice, presented in section 5.1. RIPE was compiled with this gcc for the Ubuntu 9.10 evaluation.

6 Empirical Evaluation Results

In this section, we present the summary of our empirical evaluation of the protection tools and techniques presented in section 5. We then continue with detailed evaluation results for the top four. Full log files and test results will be published online when the study is presented. The summary of the empirical evaluation is presented in table 1. Our base-case is Ubuntu 6.06, a Linux distribution released in 2006 with no countermeasures against code-injection attacks.

Setup	**Overall effectiveness**	Successful attacks	Partly successful attacks	Failed attacks
Ubuntu 6.06 (no protection)	**0%**	838 (99%)	12 (1%)	0 (0%)
Libsafe (lib wrapper)	**7%**	777 (91%)	16 (2%)	57 (7%)
LibsafePlus (lib wrapper)	**19%**	669 (79%)	16 (2%)	165 (19%)
StackShield (copy & check)	**36%**	533 (63%)	7 (1%)	310 (36%)
ProPolice (canary-based)	**40%**	501 (59%)	9 (1%)	340 (40%)
LibsafePlus+TIED (lib wrapper)	**77%**	170 (20%)	23 (3%)	657 (77%)
CRED (boundary checking)	**79%**	172 (20%)	4 (0.5%)	674 (79%)
Non-exec + stack prot (Ubuntu 9.10)	**89%**	80 (9%)	10 (1%)	760 (89%)

Table 1: **Summary of empirical evaluation results using RIPE's 850 buffer overflow attack forms. Overall effectiveness is percent of attack forms prevented. Successful attacks give repeatable arbitrary code execution. Partly successful attacks are sometimes successful, sometimes not and in general less stable. Failed attacks are repeatably prevented.**

6.1 Details for ProPolice

ProPolice is totally focused on protecting the stack and is successful in doing so for direct, stack-based buffer overflows except for structs with a buffer and function pointer. Also indirect, stack-based attacks are prevented because of the re-arranging of character buffers.

On the heap, BSS, and data segment ProPolice does not add any protective countermeasures so direct or indirect, heap/BSS/data-based attacks targeting any of the code pointers and abusing any of the functions will be successful. Indirect, heap/BSS/data-based attacks against longjmp buffers as stack variables or function parameters were not fully stable and thus categorized as partly successful.

6.2 Details for LibsafePlus + TIED

Libsafe's basic protection scheme is wrapping library functions (see list in section 5.3. This means that the only stable, successful attack forms were the ones abusing `memcpy()` and RIPE's "homebrew" memcpy equivalent since they are not wrapped.

Direct and indirect, stack/heap/BSS/data-based attacks targeting all code pointers are successful as long as they abuse `memcpy()` or RIPE's "homebrew" memcpy equivalent.

`snprintf()`, `sscanf()`, `strncpy()`, `strncat()`, `sscanf()`, `strcpy()`, `strcat()`, `fscanf()`, and `sprintf()` all were successfully abused a few times and therefore categorized as partly successful. Those partly successful attacks forms were spread across almost all other variables—direct and indirect, stack/BSS/data segment, injected code and return-into-libc and targeting return pointer, longjmp buffers, function pointers, old base pointer and structs.

6.3 Details for CRED

CRED fails to prevent direct and indirect, stack/BSS/data-based overflows toward function pointers, longjmp buffers, and structs for the library functions `sprintf()`, `snprintf()`, `sscanf()`, and `fscanf()`. The attacks against structs are also successful for `memcpy()` and homebrew memcpy equivalent and are the only attacks successful from buffers on the heap. The exception to the above is indirect attacks from the BSS and data segments targeting a longjmp buffer as stack variable. There was some instability in those attacks and therefore they were categorized as partly successful.

6.4 Details for Non-Executable Memory and Stack Protector (Ubuntu 9.10)

Ubuntu 9.10 with non-executable memory and stack protection scored the best in our evaluation. All attack forms that involved the injection of new code in a process' address space failed, due to the policy that a memory page can be either writable, or executable but not both. Also, any attack against the return address of a function was blocked due to the presence of a canary and the re-ordering of variables done by ProPolice. Strackx et. al [200] have shown cases where an attack against the stack is possible even in the presence of canary-based countermeasures, however we decided not to include such an attack in the current version.

All struct attack forms were successful meaning all locations and all abused functions worked, verifying the limitations of ProPolice. Additionally all direct attacks against function pointers on the heap and the data segment were successful. Indirect attacks against the old base pointer works in general on the heap, BSS, and data segment for `memcpy()`, `strcpy()`, `strncpy()`, `sprintf()`, `snprintf()`, `strcat()`, `strncat()`, `sscanf()`, `fscanf()`, and homebrew memcpy equivalent. But there were some instability for 10 of those combinations.

6.5 A Note on Evaluation of StackShield

The testbed execution for StackShield strangely claims only 1810 impossible attack forms whereas all the others say 2990. We figure this is because of StackShield's transformations and have manually removed the missing 1180 impossible attack forms from StackShields failed attacks since successful and partly successful attacks are obviously possible. If in fact StackShield's transformation makes 1180 extra attack forms possible, that means an increased attack surface and not enhanced protection.

6.6 Potential Shortcomings

Synthesized vs Real-World Code Testbeds

RIPE is a synthesized testbed, deliberately vulnerable, and a program with the sole purpose of conducting attacks against itself and recording their success or failure. Compared to real-world code testbeds, RIPE offers no evaluation of scalability, complexity, or performance. We see merits in both approaches—the main one for synthesized testbeds being the possibility to enumerate and combine attack forms to provide good coverage.

False Negatives and Result Manipulation

The kind of evaluation RIPE provides is susceptible to both false negatives and manipulation. An evaluated tool can prevent RIPE's implementation of a given attack form but still allow for exploitations of such attack forms in general. RIPE only provides one vulnerability and matching payload for each of the 850 attack forms, whereas in theory there are infinite variations of both. Such a case could be interpreted as a false negative. Therefore, evaluation results should be interpreted as an upper bound on the preventive effectiveness for the RIPE attack forms—there might be further successful attack forms among the 850.

Further more, researchers could inspect or observe the specifics of how RIPE implements certain attack forms and adjust their countermeasures to prevent exactly those attacks. While this could be based on bad intentions and effectively be result manipulation, it doesn't have to be. Such RIPE-specific prevention might evolve over time when fine tuning to give good evaluation results. Therefore, care has to be taken when comparing RIPE evaluation outcomes between countermeasures. We believe that it is in every researcher's own interest to use RIPE to evaluate fairly and since RIPE is free software any necessary testbed augments can be implemented and published.

7 Related Work

Pincus and Baker present an overview of recent advances in exploitation of buffer overflows [203]. Their main conclusion is that often heard assumptions about buffer overflows are not true—buffer overflows do not all inject code, do not all target the return address, and do not all abuse buffers on the stack. The article briefly discusses (1) injection of attack parameters instead of attack code, (2) attacks targeting function pointers, data pointers, exception handlers, and pointers to virtual function tables in C++, and (3) heap-based overflows.

Michael Zhivich *et al* published “Dynamic Buffer Overflow Detection” in 2005 [185]. They use a collection of 55 small, synthesized C programs that contain buffer overflows to evaluate. They have several “dimensions” in their testbed. They differentiate between *discrete* overflows of up to 8 bytes of memory, and *continuous* overflows resulting from multiple consecutive writes. They have several buffer types—`char`, `int`, `float`, `func *`, and `char *` and they are spread in the same four memory locations as we have; stack, heap, BSS, and data segment. They have buffers in struct, array, union, and array of structs. Lib functions they abuse are `(f)gets`, `(fs)scanf`, `fread`, `fwrite`, `sprintf`, `str(n)cpy`, `str(n)cat`, and `memcpy`. An attack is judged as prevented if it’s detected or if a segmentation fault occurs. The top performing tools in their study are Insure++, CCured and CRED. They also evaluate the tools against approximately 100 line models of fourteen historic vulnerabilities in bind, sendmail, and wu-ftpd. CCured, CRED, and TinyCC came out on top, detecting about 90% of the overflows. Unfortunately their testbed is not available which means their study cannot be repeated and their test cases cannot be used for future evaluations. Also they do not try all possible attack combinations nor publish exactly which buffer overflows worked and which didn’t. In contrast, RIPE is meant to be a publicly available evaluator which researchers can use to report and compare the coverage of their security mechanisms against a large but well-defined set of real-world attacks.

8 Future Work

As mentioned in previous sections, RIPE is a process that attacks itself and then checks the success or failure of the launched attacks. Due to the fact that the attack code is part of the vulnerable process, any countermeasures relying on the secrecy of memory locations are defeated since RIPE has access to the addresses of both the overflowing buffer and the target. RIPE could be extended with a *save/load offsets* feature allowing offsets from one execution to be used in a subsequent run. This would allow the evaluation of countermeasures that rely on memory randomization such as ASLR or DieHard [194] and DieHarder [195]. Heap spraying and information leakage attacks could also be added to "assist" an attacker in de-randomizing certain countermeasures. We are also currently considering the addition of non-control data attacks [204] which would allow for evaluation of countermeasures such as *data space randomization*[205] and ValueGuard [206].

9 Conclusions

Even though hundreds of papers have been published on the problem of buffer overflows and code injection attacks, modern software still is plagued by improper checking of user input attesting to the fact that this is still an open research problem. In this paper we presented RIPE, a Runtime Intrusion Prevention Evaluator which executes a total of 850 buffer-overflow attacks against popular defense mechanisms. The main purpose of RIPE is to provide a freely available testbed which developers of defense mechanisms can use to quantify the security coverage of their proposals and compare themselves against previous work using a well-defined and real-world set of attacks.

Acknowledgements: This research is partially funded by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government, the board of education and the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the IBBT and the Research Fund K.U.Leuven.

Initial parts of the testbed extensions were built by Pontus Viking as part of his Master's Thesis [25].

We are grateful to the readers who have previewed and improved our paper, especially Martin Johns.

10 Availability

RIPE is free software released under the MIT licence and available on GitHub: `https://github.com/johnwilander/RIPE`

Appendices

Appendix A

Static Testbed for Intrusion Prevention Tools

In this appendix we have included the 44 function calls used to compare publicly available tools for static intrusion prevention. To shorten it down we have only included the interesting parts.

```
#define BUFSIZE 9
static char static_global_buffer = 'A';
static char global_buffer[BUFSIZE];

/***** Buffer Overflow Vulnerabilities *****/

pointer = gets(buffer); /* Unsafe */

scanf("%8s", buffer_safe);  /* Safe */
scanf("%s", buffer_unsafe); /* Unsafe */

fscanf(fopen(file_name, "w"), "%8s", buffer_safe);  /* Safe */
fscanf(fopen(file_name, "w"), "%s", buffer_unsafe); /* Unsafe */

sscanf(input_string, "%8s", buffer_safe);  /* Safe */
sscanf(input_string, "%s", buffer_unsafe); /* Unsafe */

if(choice==0) vscanf("%8s", arglist); /* Safe */
else vscanf("%s", arglist);           /* Unsafe */

if(choice==0) vsscanf(input_string, "%8s", arglist); /* Safe */
else vsscanf(input_string, "%s", arglist);           /* Unsafe */

if(choice==0)
  vfscanf(fopen(file_name, "w"), "%8s", arglist); /* Safe */
else
  vfscanf(fopen(file_name, "w"), "%s", arglist);  /* Unsafe */

sprintf(buffer_safe, "%8s", input_string);  /* Safe */
sprintf(buffer_unsafe, "%s", input_string); /* Unsafe */

if(strlen(input_string)<BUFSIZE)
  strcat(buffer_safe, input_string);   /* Safe */
strcat(buffer_unsafe, input_string);   /* Unsafe */

if(strlen(input_string)<BUFSIZE)
  strcpy(buffer_safe, input_string);   /* Safe */
strcpy(buffer_unsafe, input_string);   /* Unsafe */

cuserid(buffer_unsafe); /* Unsafe */

if(choice==0) vsprintf (buffer_safe, "%8s", arglist); /* Safe */
else vsprintf (buffer_unsafe, "%s", arglist);         /* Unsafe */

res = streadd(buffer_safe, "a", "");            /* Safe */
res = streadd(buffer_unsafe, input_string, ""); /* Unsafe */

res = strecpy(buffer_safe, "a", "");            /* Safe */
res = strecpy(buffer_unsafe, input_string, ""); /* Unsafe */

res = strtrns("a", "a", "A", buffer_safe);            /* Safe */
res = strtrns(input_string, "a", "A", buffer_unsafe); /* Unsafe */

/***** Format String Vulnerabilities *****/

printf(&static_global_buffer); /* Safe */
printf(global_buffer);         /* Unsafe */

fprintf(stdout, &static_global_buffer); /* Safe */
fprintf(stdout, global_buffer);         /* Unsafe */
```

```
char local_buffer[BUFSIZE];
/* Safe */
sprintf(local_buffer, &static_global_buffer, input_string);
/* Unsafe */
sprintf(local_buffer, global_buffer, input_string);

char local_buffer[BUFSIZE];
/* Safe */
snprintf(local_buffer, BUFSIZE, &static_global_buffer, input_string);
/* Unsafe */
snprintf(local_buffer, BUFSIZE, global_buffer, input_string);

if(choice==0) vprintf(&static_global_buffer, arglist); /* Safe */
else vprintf(global_buffer, arglist);                  /* Unsafe */

if(choice==0) /* Safe */
  vfprintf(stdout, &static_global_buffer, arglist);
else /* Unsafe */
  vfprintf(stdout, global_buffer, arglist);

char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsprintf(local_buffer, &static_global_buffer, arglist);
else /* Unsafe */
  vsprintf(local_buffer, global_buffer, arglist);

char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsnprintf(local_buffer, BUFSIZE, &static_global_buffer, arglist);
else /* Unsafe */
  vsnprintf(local_buffer, BUFSIZE, global_buffer, arglist);
```

Appendix B

Empirical Test of Dynamic Buffer Overflow Prevention

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Halted	Halted	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 1: **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Abnormal	Prevented	Missed
Libsafe and Libverify	Halted	Missed	Halted

Table 2: **(Continued) Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 3: **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Abnormal	Missed

Table 4: **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Missed	Missed

Table 5: **(Continued) Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Abnormal	Missed
Stack Shield Global Ret Stack	Prevented	Abnormal	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 6: **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Attack Target Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 7: **(Continued) Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Appendix C

Theoretical Test of Dynamic Buffer Overflow Prevention

Attack Target Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Halted	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Missed	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Halted	Halted	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 1: **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Missed	Halted	Missed
Libsafe and Libverify	Halted	Missed	Halted

Table 2: **(Continued) Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 3: **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Missed	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 4: **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Missed	Missed

Table 5: **(Continued) Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Halted	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Missed	Halted	Missed
Libsafe and Libverify	Halted	Halted	Missed

Table 6: **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 7: **(Continued) Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Appendix D

Terminology

Specific terminology used in each of our chapters is presented in those chapters respectively. Additionally this thesis makes use of the following general terminology.

Security. The term security in computer science is related to *Computer Security* and *Information Security.* The objective of computer and information security is to protect computer services or sensitive information from data theft (ensure *Confidentiality*), data corruption (ensure *Integrity*), denial of service (ensure *Availability*) but also uphold data authenticity (ensure *Non-Repudiation*) and traceability (allow for *Auditing*). In this thesis we mostly refer to security in the sense of withstanding active intrusion attempts against benign software.

Software Security or Secure Software. We use the terms secure software and software security to denote software that has gone through some kind of security audit or test and has no known security vulnerabilities. However, there is no absolutely *secure* software since new software vulnerability types are being discovered continuously. "Secure to the best of our knowledge" would be a more accurate description. Some use the term *securish* to point out the absence of absolute security.

Security Vulnerability. A piece of software has a security vulnerability when an attacker can violate the software's security by using an attack vector. An attack vector can be malicious user input that alters or adds to the normal execution of the software but also a side channel such as measuring time, power consumption, or radiation.

Intrusion. An intrusion into benign software is made when a vulnerability is exploited to make the benign software do malicious things that violates its security.

KLOC. KLOC is short for kilo lines of code, or thousand lines of code.

Bibliography

[1] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[2] The MITRE Corporation. About cve identifiers. `http://cve.mitre.org/cve/identifiers/`.

[3] The MITRE Corporation. Cve numbering authorities. `http://cve.mitre.org/cve/cna.html`.

[4] Nist national vulnerability database – cve and cce statistics query page. `http://web.nvd.nist.gov/view/vuln/statistics`.

[5] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[6] Dennis Fisher (Threat Post). Corona ios jailbreak tool released. `http://threatpost.com/en_us/blogs/corona-ios-jailbreak-tool-released-010312`, January 2012.

[7] Lawrence R. Halme and R. Kenneth Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. `http://www.sans.org/security-resources/idfaq/aint.php`, April 2000.

[8] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, June 2006.

[9] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[10] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*. Springer-Verlag, September 2005.

[11] Martin Johns and Moritz Jodeit. Scanstud: A methodology for systematic, fine-grained evaluation of static analysis tools. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 523–530, Berlin, Germany, March 2011. IEEE.

[12] Grammatech Inc. Codesurfer. `http://www.grammatech.com/products/codesurfer/`.

[13] Pia Fåk. Modeling and pattern matching security properties with dependence graphs. Master's thesis, Linkopings universitet, August 2005.

[14] Miguel Castro. Securing software by enforcing data-flow integrity. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[15] James Clause, Wanchun Li, and Ro Orso. Dytan: A generic dynamic taint analysis framework. In *in Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[16] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms? In *in Proceedings of Third Workshop on Hot Topics in Networks, HotNets-III*, San Diego, CA USA, November 2004.

[17] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. *Microarchitecture, IEEE/ACM International Symposium on*, 0:135–148, 2006.

[18] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of The 11th Annual Network and Distributed System Security Symposium*, San Diego, USA, February 2004.

[19] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *in Proceedings of the 37th Intl Symposium on Microarchitecture, MICRO 04*, pages 209–220, 2004.

[20] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *In Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, pages 133–147, 2005.

[21] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of The 10th ACM Conference on Computer and Communications Security*, pages 272–280, Washington D.C., USA, 2003.

[22] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proceedings of The 17th Large Installation Systems Administration Conference*, San Diego, USA, October 2003.

[23] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *Security & Privacy, IEEE*, 3(6):41–49, November–December 2005.

[24] Peter Silberman and Richard Johnson. A comparison of buffer overflow prevention implementations and

weaknesses. Black Hat USA 2004 Briefings & Training http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf, 2004.

[25] Pontus Viking. Comparison of dynamic buffer overflow protection tools. Master's thesis, Linkopings universitet, February 2006.

[26] Wikipedia. Fagan inspection. http://en.wikipedia.org/wiki/Fagan_inspection.

[27] Wikipedia. Semi-structured interview. http://en.wikipedia.org/wiki/Semi-structured_interview.

[28] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[29] National Institute of Standards and Technology. Source code security analyzers. url-http://samate.nist.gov/index.php/Sourceodeecuritynalyzers.html.

[30] National Institute of Standards and Technology. The third static analysis tool exposition (sate 2010). http://samate.nist.gov/docs/NIST_Special_Publication_500-283.pdf, 2011.

[31] The MITRE Corporation. Common vulnerabilities and exposures. http://cve.mitre.org/.

[32] Jay-Evan J. Tevis and John A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 197–202, New York, NY, USA, 2004. ACM.

[33] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Software Engineering Notes*, 29:97–106, October 2004.

[34] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[35] Kendra June Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Master's thesis, Harvard University, March 2005.

[36] Kendra Kratkiewicz and Richard Lippmann. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In *E-Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools* *http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/62-kratkiewicz.pdf*, June 2005.

[37] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4), April 2006.

[38] F. Michaud and R. Carbone. Practical verification & safeguard tools for c/c++. Technical Report TR 2006-735, Defence R&D Canada – Valcartier, November 2007.

[39] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88, New York, NY, USA, 2008. ACM.

[40] James A. Kupsch and Barton P. Miller. Manual vs. automated vulnerability assessment: A case study. In *Proceedings of The 1st International Workshop on Managing Insider Security Threats (MIST 2009)* *http://ceur-ws.org/Vol-469/*, pages 83–97, June 2009.

[41] Brian Chess and Jacob West. *Secure Programming with Static Analysis.* Addison-Wesley, July 2007.

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

[43] Csaba Nagy and Spiros Mancoridis. Static security analysis based on input-related software faults. In *Proceedings of the 2009 European*

Conference on Software Maintenance and Reengineering, pages 37–46, Washington, DC, USA, 2009. IEEE Computer Society.

[44] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 25–34. IEEE, 2008.

[45] Wikipedia. Model checking. `http://en.wikipedia.org/wiki/Model_checking`.

[46] ACM. Turing award honors founders of automatic verification technology. `http://www.acm.org/press-room/news-releases-2008/turing-award-07/`.

[47] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference ACSAC*, pages 10–22, Tucson, AZ, USA, Decemeber 2005.

[48] Dawson Engler and Madanlal Musuvathi. Static analysis versus model checking for bug finding. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*. Springer, January 2004.

[49] P.Y.A. Ryan, S.A. Schneider, M.H. Goldsmith, G. Lowe, and A.W. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Pearson Education, 2000.

[50] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murphi. In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, May 1997.

[51] H Chen and D Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington DC, USA, 2002.

[52] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 389–392. ACM, 2007.

[53] Inger Anne Tøndel, Martin Gilje Jaatun, and Per Håkon Meland. Security requirements for the rest of us: A survey. *IEEE Software*, 25:20–27, January/February 2008.

[54] Wikipedia, clickjacking. `http://en.wikipedia.org/wiki/Clickjacking`.

[55] Cert® advisory ca-2000-02 malicious html tags embedded in client web requests. `http://www.cert.org/advisories/CA-2000-02.html`, February 2000.

[56] Jeff Williams, Dave Wichers, Mike Boberski, Juan Carlos Calderon, Michael Coates, Jeremiah Grossman, Jim Manico, Paul Petefish, Eric Sheridan, Neil Smithline, Andrew van der Stock, Colin Watson, OWASP Denmark Chapter (Led by Ulf Munkedal), and OWASP Sweden Chapter (Led by John Wilander). OWASP top 10 application security risks - 2010. `https://www.owasp.org/index.php/Top_10_2010-Main`, 2010.

[57] PCI Security Standards. Payment card industry data security standard (pci dss). `https://www.pcisecuritystandards.org/security_standards/documents.php?document=pci_dss_v2-0#pci_dss_v2-0`.

[58] Scott & Scott. State data breach notification laws. `http://www.scottandscottllp.com/resources/state_data_breach_notification_law.pdf`, September 2007.

[59] Wikipedia, security breach notification laws. `http://en.wikipedia.org/wiki/Security_breach_notification_laws`.

[60] Bruce Schneier. Attack trees. *Dr. Dobb's Journal*, 24(12), December 1999.

[61] Threat modeling web applications. http://msdn.microsoft.com/en-us/library/ff648006.aspx, May 2005.

[62] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7, November 1996.

[63] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[64] Vasilis Pappas. kBouncer: Effcient and transparent ROP mitigation. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.

[65] Microsoft. The bluehat prize winners announced. http://www.microsoft.com/security/bluehatprize/, July 2012.

[66] Michael Coates, John Melton, Colin Watson, Ryan Barnett, Simon Bennetts, August Detlefsen, Dennis Groves, Randy Janida, Jim Manico, Giri Nambari, Eric Sheridan, John Stevens, and Kevin Wall. OWASP appsensor project - detect and respond to attacks from within the application. https://www.owasp.org/index.php/OWASP_AppSensor_Project.

[67] CERT Coordination Center. CERT/CC statistics 1988-2004. http://www.cert.org/stats/cert_stats.html, January 2005.

[68] CSO magazine, U.S. Secret Service, and CERT Coordination Center. 2004 e-crime watch survey. http://www.csoonline.com/releases/052004129_release.html, May 2004.

[69] John Viega and Gary McGraw. *Building Secure Software : How to Avoid Security Problems the Right Way.* Addison–Wesley, 2001.

[70] Janet Burge and Dave Brown. NFRs: Fact or fiction? Computer Science Technical Report, Worcester Polytechnic Institute, WPI-CS-TR-02-01 ftp://ftp.cs.wpi.edu/pub/techreports/pdf/02-01.pdf, November 2002.

[71] Lawrence Chung, Brian A. Nixon, and Eric Yu. Using quality requirements to systematically develop quality software. In *Proceedings of the Fourth International Conference on Software Quality*, McLean, VA, USA, October 1994.

[72] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[73] Premkumar T. Devanbu and Stuart Stubblebine. Security and software engineering: A roadmap. In *Proceedings of the Twenty-second International Conference on Software Engineering, ICSE*, Limerick, Ireland, June 2000.

[74] IEEE. IEEE-STD 610.12-1990, IEEE standard glossary of software engineering terminology, May 1990.

[75] Richard H. Thayer and Merlin Dorfman. *Software Requirements Engineering, Second Edition*. IEEE Computer Society Press and John Wiley & Sons, Inc., 1999.

[76] National Institute of Standards and Technology. Common criteria for information technology security evaluation (CC 2.1). `http://csrc.nist.gov/cc/CC-v2.1.html`.

[77] International Organization for Standardization. ISO/IEC 17799:2000 information technology – code of practice for information security management. `http://www.iso.org/iso/en/prods-services/popstds/informationsecurity.html`.

[78] Herbert H. Thompson and James A. Whittaker. Testing for software security. *Dr. Dobb's Journal*, 27(11):24–32, November 2002.

[79] Microsoft. Microsoft security glossary. `http://www.microsoft.com/security/glossary.mspx`, November 2004.

[80] Robert W. Shirey. Request for comments: 2828, Internet security glossary. `http://www.faqs.org/rfcs/rfc2828.html`, May 2000.

[81] European Union. Common procurement vocabulary. `http://europa.eu.int/scadplus/leg/en/lvb/l22008.htm`, 2004.

[82] Mercell AB Sweden. Database of all purchases in the category '72 - computer and related services' made by swedish government or local authorities from January 2003 to June 2004. `http://www.mercell.com`, 2004.

[83] John Wilander and Jens Gustavsson. Security requirements—a field study of current practice. In *E-Proceedings of the Symposium on Requirements Engineering for Information Security, in conjunction with the 13th IEEE International Requirements Engineering Conference (to appear in formal proceedings)*, Paris, France, August 2005.

[84] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

[85] Albert Alderson. False requirements express real needs. *Requirements Engineering*, 4:60–61, May 1999.

[86] Ross J. Anderson. *Security Engineering—A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

[87] John McDermott and Chris Fox. Using abuse case models for security requirements analysis. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC'99)*, Scottsdale, AZ, USA, December 1999.

[88] Ian Alexander. Initial industrial experience of misuse cases in trade-off analysis. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 61–68, Essen, Germany, September 2002.

[89] Donald G. Firesmith. Engineering security requirements. *Journal of Object Technology*, 2(1):53–68, January–February 2003.

[90] Donald G. Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3(1):61–75, January-February 2004.

[91] Guttorm Sindre, Donald G. Firesmith, and Andreas L. Opdahl. A reuse-based approach to determining security requirements. In *Proceedings of the 9th International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'03)*, Klagenfurt/Velden, Austria, June 2003.

[92] Lin Liu, Eric Yu, and John Mulopoulos. Security and privacy requirements analysis within a social setting. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*, pages 151–161, Monterey Bay, CA, USA, September 2003.

[93] Lin Liu, Eric Yu, and John Mylopoulos. Analyzing security requirements as relationships among strategic actors. In *Proceedings of the 2nd Symposium on Requirements Engineering for Information Security (SREIS'02)*, Raleigh, North Carolina, USA, October 2002.

[94] Eric Yu and Luiz Marcio Cysneiros. Designing for privacy and other competing rerquirements. In *Proceedings of the 2nd Symposium on Requirements Engineering for Information Security (SREIS'02)*, Raleigh, North Carolina, USA, October 2002.

[95] Ivar Jacobson. Object oriented development in an industrial environment. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'87)*, pages 183–191, Orlando, Florida, USA, 1987.

[96] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, December 2003.

[97] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements by misuse cases. In *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, Sydney, Australia, November 2000.

[98] Ian Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 20:58–66, January/February 2003.

[99] Luncheng Lin, Bashar Nuseibeh, Darrel Ince, Michael Jackson, and Jonathan Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings of 11th International IEEE Requirements Engineering Conference (RE'03)*, pages 371–372, Monterey Bay, CA, USA, September 2003.

[100] Robert Crook, Darrel Ince, Luncheng Lin, and Bashar Nuseibeh. Security requirements engineering: When anti-requirements hit the fan. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE'02)*, Essen, Germany, September 2002.

[101] Michael Howard and David LeBlanc. *Writing Secure Code, 2nd Edition.* Microsoft Press, 2002.

[102] CERT Coordination Center. Cert/cc statistics 1988-2001. `http://www.cert.org/stats/`, February 2002.

[103] Computer Science and National Research Council Telecommunications Board. Cybersecurity today and tomorrow: Pay now or pay later (prepublication). Technical report, National Academies, USA, `http://www.nap.edu/books/0309083125/html/`, January 2002.

[104] Lisa M. Bowman. Companies on the hook for security. `http://news.com.com/2100-1023-821266.html`, January 2002.

[105] BBC News. Software security law call. `http://news.bbc.co.uk/hi/english/sci/tech/newsid_1762000/1762261.stm`, January 2002.

[106] Anup K. Ghosh, Chuck Howell, and James A. Whittaker. Building software securely from the ground up. *IEEE Software*, 19(1):14–16, February 2002.

[107] John Wilander. Security intrusions and intrusion prevention. Master's thesis, Linkopings universitet, `http://www.ida.liu.se/~johwi`, April 2002.

[108] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.

[109] Gary McGraw and John Viega. An analysis of how buffer overflow attacks work. IBM developerWorks: Security: Security articles http://www-106.ibm.com/developerworks/security/library/smash.html?dwzone=security, March 2000.

[110] DilDog. The tao of Windows buffer overflow. http://www.cultdeadcow.com/cDc_files/cDc-351/, April 1998.

[111] Matt Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, January 1999.

[112] David A. Wheeler. Secure programming for Linux and Unix HOWTO v2.89. http://www.dwheeler.com/secure-programs/, October 2001.

[113] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[114] tf8. Bugtraq id 1387, Wu-Ftpd remote format string stack overwrite vulnerability. http://www.securityfocus.com/bid/1387, June 2000.

[115] Scut and Team Teso. Exploiting format string vulnerabilities. http://teso.scene.at/articles/formatstring/, September 2001.

[116] Tim Newsham. Format string attacks. White Paper http://www.guardent.com/rd_whtpr_formatNewsham.html, September 2000.

[117] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. White Paper http://www.research.avayalabs.com/project/libsafe/, December 1999.

[118] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[119] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, Spring 1996.

[120] David A. Wheeler. Flawfinder. Web page `http://www.dwheeler.com/flawfinder/`, May 2001.

[121] Secure Software Soliutions. Rough auditing tool for security, RATS 1.3. `http://www.securesw.com/rats/`, September 2001.

[122] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[123] C E Pramode and C E Gopakumar. Static checking of C programs with LCLint. *Linux Gazette*, 51 `http://www.linuxgazette.com/issue51/pramode.html`, March 2000.

[124] S. C. Johnson. Lint, a C program checker. AT&T Bell Laboratories: Murray Hill, NJ. `http://citeseer.nj.nec.com/johnson78lint.html`, July 1978.

[125] Jose Nazario. Project pedantic—source code analysis tool(s). `http://pedantic.sourceforge.net/`, March 2002.

[126] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, 1997.

[127] Anup Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, May 1998.

[128] Wenliang Du and Aditya P. Mathur. Vulnerability testing of software system using fault injection. COAST, Purdue University, Technical Report 98-02 `http://www.cerias.purdue.edu/coast/coast-library.html`, April 1998.

[129] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, `http://www.cs.berkeley.edu/~ushankar/`, August 2001.

[130] Jose Nazario. Source code scanners for better code. The Linux Journal `http://www.linuxjournal.com/article.php?sid=5673`, January 2002.

[131] Pete Broadwell and Emil Ong. A comparison of static analysis and fault injection techniques for developing robust system services. Technical report, Computer Science Division, University of California, Berkeley, `http://www.cs.berkeley.edu/~pbwell/saswifi.pdf`, May 2002.

[132] John Wilander and Mariam Kamkar. A comparative study 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.

[133] J S Foster, R Johnson, J Kodumal, T Terauchi, U Shankar, K Talwar, D Wagner, A Aiken, M Elsman, and C Harrelson. Cqual: A tool for adding type qualifiers to C. `http://www.cs.umd.edu/~jfoster/cqual/`, 2003.

[134] W Chen, B Rudiak-Gould, and B Schwartz. Automatic detection of implicit type cast errors in C. Paper in graduate course, `http://www.cs.berkeley.edu/~wychen/papers/261.ps`, 2002.

[135] R Johnson and D Wagner. Checking linux kernel user-space pointer handling with cqual. Work-in-progress report at IEEE Symposium on Security and Privacy, May 2003.

[136] K Ashcraft and D Engler. Using programmer written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[137] V B Livshits and M S Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the*

11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Helsinki, Finland, 2003.

[138] Steven S Muchnick. *Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[139] M Weber, V Shah, and C Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[140] B V Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[141] B V Chess. Personal communication, 2004.

[142] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[143] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[144] M Musuvathi and D Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of the Second Workshop on Software Model Checking*, Boulder, Colorado, USA, 2003.

[145] K J Ottenstein and L M Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177—184, Pittsburg, Pennsylvania, 1984.

[146] N Walkinshaw, M Wood, and M Roper. The java system depencence graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, 2003.

[147] S Horwitz, T Reps, and D Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[148] J Ferrante, K J Ottenstein, and J D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[149] M Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, USA, 1981.

[150] T Reps and G Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, Washington DC, USA, 1995.

[151] Blexim. Basic integer overflows. Phrack Magazine 60 `http://www.phrack.org/phrack/60/p60-0x0a`, 2002.

[152] M Howard. Reviewing code for integer manipulation vulnerabilities. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp`, April 2003.

[153] David A. Wheeler. Secure programming for Linux and Unix HOWTO v3.010. `http://www.dwheeler.com/secure-programs/`, March 2003.

[154] John Wilander. Modeling and visualizing security properties of code using dependence graphs. In *Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden (to appear)*, Vasteras, Sweden, `http://www.idt.mdh.se/serps-05/`, October 2005.

[155] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, 1979.

[156] Frederick Giasson. Memory layout in program execution. `http://www.decatomb.com/articles/memorylayout.txt`, October 2001.

[157] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*, pages 119–129, Hilton Head, South Carolina, January 2000.

[158] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[159] Tzi cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

[160] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[161] Bulba and Kil3r. Bypassing StackGuard and StackShield. Phrack Magazine Volume 10, Issue 56 `http://www.phrack.org/phrack/56/p56-0x05`, May 2000.

[162] Crispin Cowan. Personal communication, February 2002.

[163] Vendicator. Stack Shield technical info file v0.7. `http://www.angelfire.com/sk/stackshield/`, January 2001.

[164] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[165] Istvan Simon. A comparative analysis of methods of defense against buffer overflow attacks. `http://www.mcs.csuhayward.edu/~simon/security/boflo.html`, January 2001.

[166] Mike Shuey Mike Frantzen. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[167] George Necula, Scott McPeak, and Wes Weimer. Taming C pointers. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, June 2002.

[168] George Necula, Scott McPeak, and Wes Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[169] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[170] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging AADEBUG'97*, Linkoping, Sweden, May 1997.

[171] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with StackGuard. Linux Expo `http://www.cse.ogi.edu/~crispin/`, May 1999.

[172] Solar Designer. Linux kernel patch from the openwall project. `http://www.openwall.com/linux/README`.

[173] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, May 2001.

[174] Wirex Crispin Cowan. Nearly 100 hackers fail to crack wirex immunix server, August 2002.

[175] Pierre-Alain Fayolle and Vincent Glaume. A buffer overflow study, attacks & defenses. http://www.enseirb.fr/~glaume/indexen.html, March 2002.

[176] Eugene H. Spafford and Eugene H. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19, 1988.

[177] David Moore, Colleen Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. In *2nd ACM Workshop on Internet measurment*, 2002.

[178] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[179] US-CERT. Vulnerability notes database. http://www.kb.cert.org/vuls.

[180] Standard Performance Evaluation Corporation (SPEC). Spec cpu benchmark suites. http://www.spec.org/cpu/.

[181] John Wilander and Mariam Kamkar. A comparative study of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network & Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[182] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In Ryoichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP Advances in Information and Communication Technology*, pages 375–391. Springer Boston, 2005.

[183] Michael Howard. Evils of strncat and strncpy - answers. http://blogs.msdn.com/b/michael_howard/archive/2004/12/10/279639.aspx, December 2004.

[184] Petr Gajdos and Christian Kornacker. Cve-2009-4035 xpdf: buffer overflow in fofitype1. https://bugzilla.redhat.com/show_bug.cgi?id=541614, December 2009.

[185] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. Workshop on the Evaluation of Software Defect Detection Tools, co-located with PLDI 2005 http://ewww.cs.umd.edu/~pugh/BugWorkshop05/, June 2005.

[186] Brandon Bray. Compiler security checks in depth. http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx, February 2002.

[187] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium*, San Diego, USA, February 2005.

[188] Danny Nebenzahl and Avishai Wool. Install-time vaccination of windows executables to defend against stacksmashing attacks. In *Proceedings of The 19th IFIP International Information Security Conference*, Toulouse, France, August 2004.

[189] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows. To appear in Software—Practice & Experience.

[190] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of The 13th USENIX Security Symposium*, pages 45–56, San Diego, USA, August 2004.

[191] Dionysus Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. In *BlackHat DC*, 2010.

[192] grsecurity. Pax. http://pax.grsecurity.net/.

[193] Arjan van de Ven and Ingo Molnar. Execshield. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

[194] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 conference on Programming language design and implementation*, pages 158–168, Ottawa, ON, 2006. ACM Press.

[195] Gene Novark and Emery D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[196] Ubuntu 9.10 karmic. `http://releases.ubuntu.com/karmic/`.

[197] Ubuntu security feature matrix. `https://wiki.ubuntu.com/Security/Features`.

[198] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *In Proceedings of the 11th ACM conference on Computer and communications security, CCS '04*, 2004.

[199] Tyler Durden. Bypassing pax aslr protection. `http://www.phrack.com/issues.html?issue=59&id=9`, July 2002.

[200] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, 2009.

[201] Riley Hassell and Ryan Permeh. Microsoft internet information services remote buffer overflow. `http://www.eeye.com/Resources/Security-Center/Research/Security-Advisories/AD20010618`, 6 2001.

[202] Wikipedia, nx bit. `http://en.wikipedia.org/wiki/NX_bit`.

[203] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[204] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, 2005.

[205] Sandeep Bhatkar and R. Sekar. Data space randomization. In *In Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*, July 2008.

[206] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. Valueguard: Protection of native applications against data-only buffer overflows. In Somesh Jha and Anish Mathuria, editors, *Information Systems Security*, volume 6503 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin / Heidelberg, 2011.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.