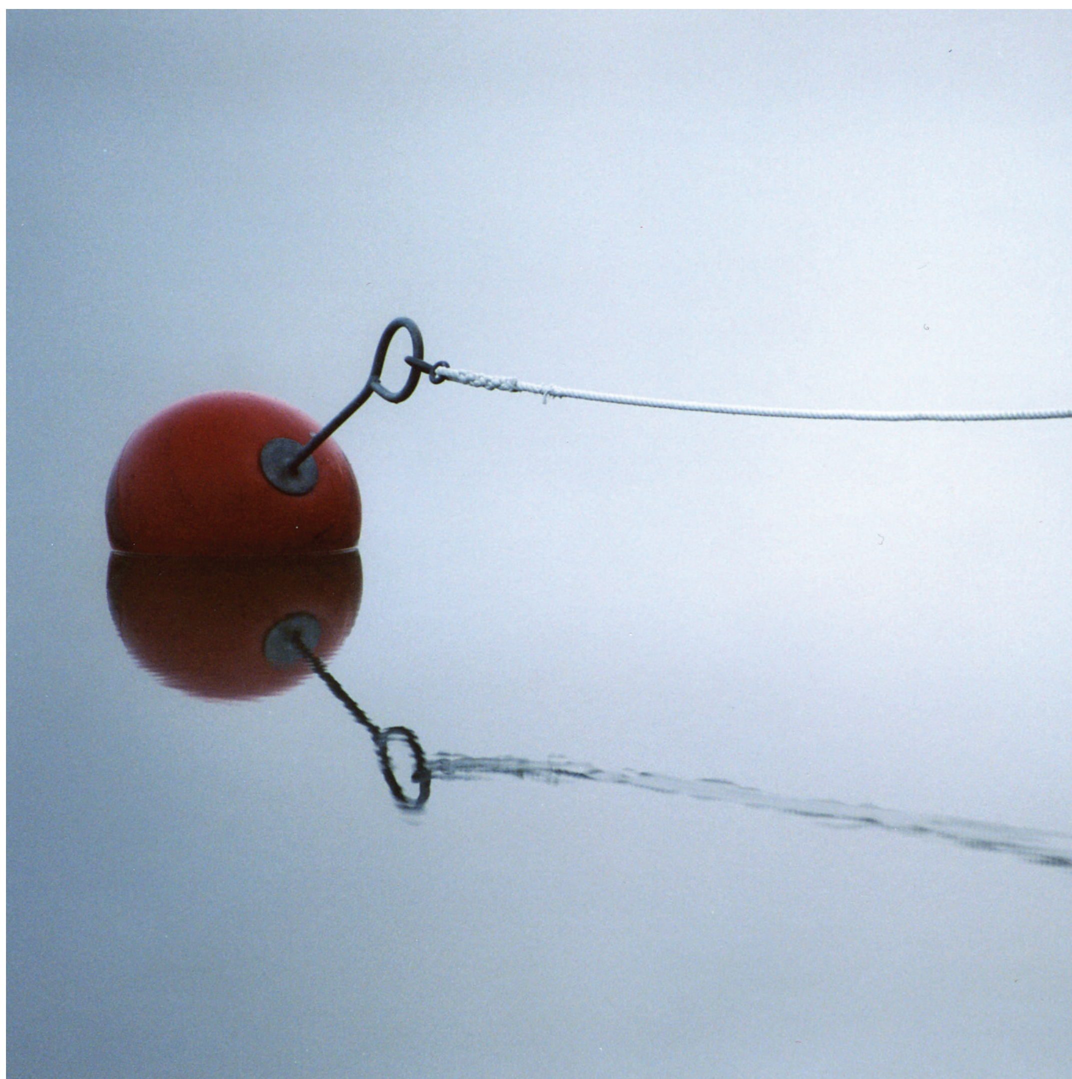
FOI

Verktyg för att åstadkomma pålitlig programvara

IOANA RODHE, MARTIN KARRESAND



FOI-R--4290--SE
ISSN 1650-1942 September 2016

Ioana Rodhe, Martin Karresand

Verktyg för att åstadkomma pålitlig programvara

Bild/cover: Martin Karresand

Titel	Verktyg för att åstadkomma pålitlig programvara
Title	Tools to accomplish trustworthy software
Rapportnummer	FOI-R--4290--SE
Månad	September
Utgivningsår	2016
Antal sidor	62
ISSN	ISSN-1650-1942
Uppdragsgivare	Försvarsmakten
Projektnummer	E36077
Godkänd av	Christian Jönsson
Ansvarig avdelning	Ledningssystem
Forskningsområde	Informationssäkerhet och kommunikation
FoT-område	Ledning och MSI

Detta verk är skyddat enligt lagen (1960:729) om upphovsrätt till litterära och konstnärliga verk. All form av kopiering, översättning eller bearbetning utan medgivande är förbjuden.

This work is protected under the Act on Copyright in Literary and Artistic Works (SFS 1960:729). Any form of reproduction, translation or modification without permission is prohibited.

Sammanfattning

Denna rapport beskriver ett urval av de allmänt tillgängliga verktyg som används för att åstadkomma pålitlig programvara, med syfte att ge en översikt av aktuell status på forskningen inom området pålitlig programvara. Urvalet är baserat på författarnas litteratursökningar under FoT-projekten Teknik för IT-säkerhet och Pålitliga IT-plattformar. Rapporten presenterar på en övergripande nivå den kunskap som förvärvats i projekten. Ett visst överlapp med material publicerat i tidigare rapporter i projektet finns därför.

De olika klasser med verktyg som berörs i rapporten är (1) formella metoder, (2) standarder, (3) organisationer, (4) säkra programspråk, (5) sårbarhetsförutsägelse och (6) säkra operativsystem. Dessa verktygsklasser presenteras översiktligt och förklaras på en grundläggande nivå. Rapporten är ett stöd för de inom Försvarsmakten som behöver kunskap om området pålitlighet i programvara.

Slutsatserna av studien är att forskning och utveckling rörande verktyg för att åstadkomma pålitlighet i programvara har ökat på senare tid, men att den ännu inte nått tillräcklig mognadsgrad för att slå igenom fullt ut i produktionsledet. Användning av standarder och certifiering sker regelmässigt. Programvaruutvecklingsverktyg baserade på formella metoder visar lovande resultat, men kräver i de flesta fall fortfarande speciallösningar för att vara användbara.

Nyckelord

pålitlig IT-komponent, pålitlighet, programvara, formell metod

Abstract

This report describes a selection of tools used to achieve trustworthy software, with the aim to give an overview of the current status of the research within the area of trustworthy software. The selection is based on the authors' literature surveys during the projects Technique for IT Security and Trustworthy IT Platforms. The report presents the knowledge attained in the projects on a comprehensive level. A certain overlap with material published in other reports in the projects is therefore present.

The different classes of tools covered in the report are (1) formal methods, (2) standards, (3) organisations, (4) secure programming languages, (5) vulnerability prediction and (6) secure operating systems. These classes of supporting tools are presented briefly and explained on a basic level. The report is a support for those in the Swedish Armed Forces that need to gain knowledge within the area of trustworthy software.

The conclusions drawn from the study are that research and development of technology giving trustworthy software have increased recently, but have not yet reached a maturity level allowing commercial use. Standards and certification are used regularly. Software development tools based on formal methods are showing promising results, but still often require special solutions to be useful.

Keywords

trustworthy IT component, trustworthiness, software, formal method

Innehåll

1 Inledning **7**
1.1 Definition . . . 8
1.2 Syfte . . . 9
1.3 Metod . . . 9
1.4 Avgränsningar . . . 10
1.5 Disposition . . . 10

2 Formella metoder **11**
2.1 Metoder för beskrivning och analys . . . 13
2.1.1 Modellbaserad specifikation . . . 14
2.1.2 Deklarativ modellering . . . 14
2.1.3 Axiomatisk specifikation . . . 15
2.2 Bevismekanismer . . . 16
2.2.1 Logik . . . 16
2.2.2 Utvecklingsverktyg för logisk validering . . . 18
2.2.3 Modellvalidering . . . 19
2.2.4 Programlogik och -kommentering . . . 19
2.3 Koppling mellan modell och program . . . 20
2.3.1 Förfining . . . 20
2.3.2 Extrahering . . . 20
2.3.3 Exekvering . . . 20
2.4 Skalbarhet . . . 21
2.5 Semi-formella metoder . . . 21

3 Standarder **23**
3.1 Common Criteria . . . 23
3.2 Standarden FIPS 140-2 . . . 25
3.3 DO-178C/ED-12C . . . 26
3.4 Administrativa standarder . . . 27
3.4.1 ISO/IEC 27000-serien . . . 27

3.4.2 BITS – Basnivå för informationssäkerhet 27
3.4.3 ISO/IEC 13335 27

4 Organisationer, initiativ och projekt för ökad pålitlighet 29
4.1 TCG 29
4.2 TSI 30
4.3 DHS SAP 30
4.4 NIST och SAMATE 31
4.5 SAFECode 32

5 Programspråk 33
5.1 Säkerhetstypade programspråk 33
5.2 Andra säkra programspråk 36
5.3 Standarder för säker programmering 36
5.4 Sårbarhetsförutsägelse i kod 38

6 Säkra operativsystem 41
6.1 SELinux 42
6.2 TrustedBSD 42
6.3 STOP OS 43
6.4 INTEGRITY-178B 43

7 Diskussion och slutsatser 45
7.1 Diskussion 45
7.2 Slutsatser 47
7.3 Framtida arbete 48

Litteratur 49

A Förkortningar 59

1 Inledning

Några av de verktyg[1] som möjliggör pålitlighet hos programvara är formella metoder, standarder, certifiering[2], checklistor och guider, säkra operativsystem och säkra programspråk. Verktygen är delvis överlappande, till exempel ingår formella metoder i vissa standarder när någonting ska bevisas med en hög pålitlighetsnivå. På samma sätt inkluderar vissa säkra programspråk formella metoder och vissa organisationer arbetar för att det ska finnas standarder som programvara enklare kan certifieras mot.

Begreppet formella metoder omfattar en mängd metoder för att bevisa riktighet för hela eller delar av utvecklingsprocessens utdata. Området är dock fortfarande under utveckling vilket gör att det finns en mängd olika kategoriseringar av formella metoder att tillgå, men inte någon allmänt vedertagen eller standardiserad sådan. Likaså är mängden programvaruutvecklingsverktyg[3] starkt skiftande mellan olika formella metoder. Ett fåtal formella metoder har anammats av den kommersiella världen och det har därför skapats fler utvecklingsverktyg för just de metoderna. I Avsnitt 2 introduceras en kategorisering av formella metoder tillsammans med en förklaring av hur och i vilka steg i utvecklingsprocessen de kan användas. Dessutom görs en översiktlig genomgång av några av de mer populära formella metoderna.

En av de mer kända standarderna som används inom den kommersiella världen för att utveckla och certifiera pålitliga system är Common Criteria for Information Technology Security Evaluation (CC). I Avsnitt 3 presenteras CC närmare, tillsammans med ett urval standarder för de delar som CC inte täcker.

Standarder och andra ramverk för höjning av programkvalitet måste underhållas och förändras i takt med att programmeringsområdet utvecklas. Detta arbete utförs av olika organisationer och frivilliga initiativ. Utöver underhåll av standarder utvecklar de även befintliga och nya programspråk för att underlätta utveckling av säker och pålitlig programvara. Några av dessa organisationer, frivilliga initiativ och programspråk avhandlas i Avsnitt 4 och 5.

En viktig del i pålitliga IT-system är användning av *säkra operativsystem*, som

[1] I rapporten kommer termen *(pålitlighetshöjande) verktyg* användas för de övergripande sätt som kan användas för att öka pålitligheten hos programvara.

[2] Certifiering innebär en standardiserad process som mäter om fördefinierade krav uppfylls enligt en given standard. De institutioner och laboratorier som får utföra certifieringar brukar också vara certifierade själva. Att vara certifierad enligt en viss standard ökar omgivningens tilltro till arbetet inom ramen för standarden.

[3] Termen *(programvaru)utvecklingsverktyg* kommer i rapporten att användas för de specifika hjälpmedel som tagits fram inom respektive kategori av pålitlighetshöjande verktyg som presenteras.

Tabell 1: FOI-rapporter och -memon som skrivits i FoT-projekten Pålitliga IT-plattformar (PLIT) och Teknik för IT-säkerhet (TIST).

År	Rapport
2013	J. Löfvenberg och I. Rodhe. *Litteraturstudie av tekniker för pålitliga IT-plattformar.* FOI-R--3724--SE.
2013	N. Lewau och J. Löfvenberg. *Pålitliga IT-system i Försvarsmakten – en behovsinventering.* FOI-R--3799--SE.
2013	M. Persson. *Pålitlig IT-plattform – Teknisk beskrivning av en demonstrator.* FOI MEMO 4723.
2013	L. Westerdahl och J. Löfvenberg. *Årsrapport 2013 – Pålitliga IT-plattformar.* FOI MEMO 4697.
2014	M. Karresand. *Pålitliga IT-plattformar – Kopplingar mellan Försvarsmaktens behov och litteraturen.* FOI-R--3903--SE.
2014	D. Eidenskog och L. Westerdahl. *Metoder för informations- och åtkomstkontroll.* FOI-R--4010--SE.
2014	A. G. Hunstad och J. Löfvenberg. *Årsrapport 2014 – Pålitliga IT-plattformar.* FOI-R--3903--SE.
2015	L. Westerdahl. *Objektbaserad säkerhet – omvärldsbevakning av attributbaserad kryptering.* FOI MEMO 5512.
2015	I. Rodhe och M. Karresand. *Overview of formal methods in software engineering.* FOI-R--4156--SE.
2015	A. G. Hunstad och I. Rodhe. *IT-säkerhets- och informationssäkerhetsutbildningar i Sverige.* FOI-R--4160--SE.

är särskilt framtagna för ändamålet. I Avsnitt 6 beskrivs de krav som ställs på ett säkert operativsystem och vilka metoder som används för att säkerställa dessa krav. Det görs även en kort genomgång av några säkra operativsystem.

Denna rapport samlar erfarenheter och sammanställer kunskap från FoT-projektet Teknik för IT-säkerhet (TIST) och till viss mån även dess föregångare Pålitliga IT-plattformar (PLIT). Delar av innehållet i rapporten har täckts ytterligare i andra FOI-rapporter, vilka visas i Tabell 1. I TIST pågår även studier av praktisk tillämpbarhet hos olika formella metoder och då särskilt *model checkers*.

1.1 Definition

Pålitlighet är ett luddigt begrepp definitionsmässigt sett. De flesta är dock någorlunda överens om vad det innebär. I IT-sammanhang finns kopplingar till begreppet säkerhet och stundtals används begreppen i stort sett synonymt. I en tidigare FOI-rapport [58, sid. 10] har pålitlighet definierats som ”*[e]genskapen att riktigheten kan attesteras*”. Riktighet definieras i samma rapport som:

> *[e]genskap eller tillstånd som innebär att information eller funktion stämmer överens med relevant specifikation och inte har ändrats, vare sig obehörigt eller av misstag.* [58, sid. 10]

Attestering, som i vardagligt tal har innebörden ”skriftligt intyga riktighet av dokument e.d.” [72], definieras i FOI-rapporten som:

> *[a]tt med en väldefinierad metod bevisa eller verifiera ett påstående om ett objekt. Vilken egenskap som avses måste specificeras på något sätt.* [58, sid. 10]

Den definition av pålitlighet som presenteras ovan och som används inom PLIT och TIST kan liknas vid principen med att lägga ett original och en kopia av samma dokument ovanpå varandra och hålla dem mot en ljus bakgrund för att se att de stämmer överens. Dokumentkopian är därmed pålitlig eftersom dess textuella innehåll (information) stämmer överens med dokumentoriginalet (relevant specifikation), vilket attesteras genom att med hjälp av genomlysning verifiera påståendet att original och kopia är visuellt överensstämmande.

1.2 Syfte

Syftet med rapporten är ge en översikt över några av de allmänt tillgängliga verktyg som åstadkommer pålitlighet i programvara. Rapporten riktar sig till de som önskar få kännedom om koncept och termer som används inom forskningsområdet pålitlighet i programvara. Även forskningsresultat från standardiserad och strukturerad programutveckling samt formella metoder tas upp till viss del. För den som vill ha en mer gedigen vetenskaplig bakgrund med värdering av de olika metoderna och relevans för Försvarsmakten hänvisas till Tabell 1 som visar övriga rapporter som skrivits inom projektet. Värdet för Försvarsmakten av denna rapport är som en introduktion till forskningsområdet pålitlighet i programvara.

1.3 Metod

Under arbetet med denna rapport tillämpades en ad hocbaserad iterativ metod för litteratursökning. Den specifika metod som valdes baserades på girighetsprincipen kombinerad med djup-först-sökning. Basen för girighetsalgoritmen var antal referenser, omnämnanden och träffbetyg i sökmotorn. Först gjordes breda sökningar på internet och vartefter kunskapen om området ökade gjordes ytterligare sökningar baserade på de nya kunskaperna. Vid varje iteration användes de av girighetsalgoritmen högst rankade träffarna.

I arbetet ingick även kontakter med inom Sverige verksamma forskargrupper på högskolor och universitet. De kontakterna gav ytterligare kunskaper och förslag på lämplig litteratur, vilka användes för förnyade informationssökningar på internet. Den samlade kunskapen från dessa sökningar ligger till grund för urvalet av

de verktyg för att åstadkomma pålitlighet i programvara som presenteras i rapporten.

1.4 Avgränsningar

Rapportens omfattning har begränsats till verktyg som av författarna uppfattas som allmänt tillgängliga, etablerade och stabila. Urvalet är således inte heltäckande, utan är ämnat att ge läsarna en överblick över området med grundläggande förklaringar av respektive verktyg. Ingen testning eller utvärdering av verktygen har gjorts och därför presenteras heller inte några resultat, rekommendationer eller slutsatser om dessa.

1.5 Disposition

Rapporten består av följande kapitel:

Kapitel 2: Kapitlet utgör en översikt av området formella metoder, klassificerat i fyra kategorier i enlighet med [4, Kap. 2]; beskrivning och analys, bevismekanismer, koppling mellan modell och program, samt skalbarhet.

Kapitel 3: Kapitlet presenterar några av de större standarderna för skapande av pålitliga system med bäring på IT-säkerhet.

Kapitel 4: Kapitlet ger en överblick över de större organisationer och projekt som arbetar med utveckling och befrämjande av metoder för pålitlighet.

Kapitel 5: Kapitlet tar upp olika varianter av säkra programspråk, sårbarhetsförutsägelse, samt standarder för säker programmering.

Kapitel 6: Kapitlet avhandlar fyra olika operativsystem, som byggts för att vara pålitliga.

Kapitel 7: Kapitlet innehåller en diskussion av de viktigaste resultaten av studien tillsammans med de slutsatser som dragits av arbetet och förslag på fortsatt arbete.

Bilaga A: Sist i rapporten, efter litteraturlistan, finns en bilaga med förklaringar av de förkortningar som används i texten.

2 Formella metoder

Det traditionella sättet att öka pålitligheten hos programvara är att utföra tester för att identifiera avvikelser från avsett beteende. Problemet med denna metodik är dock att den i praktiken aldrig kan bli heltäckande och några garantier om programvarans pålitlighet går därför inte att utställa.

Ett annat sätt att angripa pålitlighetsproblemet är att använda *formella metoder*. Dessa är matematiskt grundade och kan användas för specificering, utveckling och verifiering av både programvara och hårdvara.

I [87] beskrivs termen formella metoder som:

> *The term* formal methods *refers to the use of mathematical modeling, calculation and prediction in the specification, design, analysis and assurance of computer systems and software. The reason it is called* formal methods *rather than* mathematical modeling of software *is to highlight the character of the mathematics involved.* [87, sid. 16]

Användningen av formella metoder har ökat på senare år och börjat sprida sig utanför forskningsvärlden. En bidragande orsak till denna spridning är de ökande kraven på säkerhet i IT-system. För att hantera dessa krav används standardiserad metodik, till exempel CC, som i sin tur i vissa fall kräver användning av formella metoder. I flera fall kräver numera olika säkerhetsstandarder att formella metoder används i olika grad för de högre säkerhetsklasserna. Till exempel ingår användningen av formella metoder som ett krav i CC för att ett system ska få klassificeras som Evaluation Assurance Level (EAL) 5[1]–7 [4, sid. 4].

En fördel med formella metoder är att de går att tillämpa partiellt, det vill säga att de ger positiv effekt även om de bara används under delar av utvecklingsprocessen eller för enstaka delar i systemet. Ju större del av processen som stöds av formella metoder, desto större tilltro till att resultatet är korrekt. Det är dock extrakostnader förbundna med användningen av formella metoder, både i form av att utvecklarna måste känna sig bekväma med vald metod och att det måste finnas bra utvecklingsverktyg som stöder den. Det krävs även en högre matematisk kunskap hos de inblandade än när formella metoder inte används. På grund av de högre kraven vid användning av formella metoder tillämpas de oftast enbart på säkerhetskritiska delar av ett system.

[1] Strikt talat kräver CC användning av *semiformella metoder* för designen vid assurans på EAL-nivå 5. Semiformella metoder är välstrukturerade programutvecklingsmetoder som inte är matematiskt stringenta, men ändå ökar tilltron till att utvecklingen sker välordnat. Unified Modeling Language (UML) är ett exempel på ett semiformellt språk. Mer kan läsas i FOI-rapporten FOI-R--4156--SE.

Det har visats att användningen av formella metoder minskar de faktiska kostnaderna och ökar effektiviteten i utvecklingsprocessen [114]. Förbättringen är mer påtaglig om de formella metoderna appliceras tidigt i processen, helst redan i specifikationsfasen. Anledningen till att effekten blir störst då är att fel som ger följdfel i en lång kedja undviks, vilket annars blir kostsamt att korrigera.

Formella metoder kan till exempel användas på följande sätt:

Formell specifikation används för att med hjälp av någon formell metod omvandla kravställningar till en modell av det önskade systemet. Detta gör det möjligt att systematiskt analysera kraven ur ett konsekvens- och helhetsperspektiv. Detta steg innehåller inte några beskrivningar på detaljnivå, endast av enheterna i systemet och deras egenskaper och samverkan.

Formell design ger en formell beskrivning av planerad eller faktisk implementation av systemet.

Verifikation görs både för formell specifikation och design. För detta kan antingen bevis av teorem eller modellvalidering användas.

Huvudproblemet som användningen av formella metoder försöker lösa är hur en specifikation överförs till ett färdigt datorprogram under verifierbara former. Problemet kan i systemutvecklingsmiljö delas in i två delproblem [4, sid. 16]:

Modellvalidering Hur kan ett önskat beteende säkerställas på specifikationsnivå?

Formell koppling mellan specifikation och implementation Hur uppnås en implementation med samma beteende som preciserats i specifikationen?

Frågorna besvaras lite olika beroende på vilken klass av formella metoder som används. Några av de vanligare sätten att ta sig an modellvalidering är *animering*[2], *transformation*[3] eller *bevis av egenskaper*. Kopplingsproblemet löses oftast med hjälp av *konstruktion*[4] eller *verifikation*[5].

[2] Animering är en enklare form av exekvering där en specifikation kan provköras på rudimentär nivå. Det finns olika utvecklingsverktyg för animering av specifikationer, många har kopplingar till specifikationsspråken Z och B. I [56] förklaras animering med hjälp av språket Structure Object-oriented Formal Language (SOFL).

[3] Transformation innebär att någonting (en specifikation, modell, källkod etcetera) överförs från ett tillstånd eller format till ett annat. Kompilering av källkod till körbar kod är ett exempel.

[4] Konstruktion innebär att riktigheten hos programvaran garanteras genom en rigoröst kontrollerad utvecklingsprocess, där varje steg är formellt verifierat. Det är alltså *processen* fram till resultatet som är viktig.

[5] Vid verifikation ligger fokus på den färdiga implementationen och matematiska bevis används för att verifiera att den uppfyller specifikationen. Således är det *resultatet* av processen som är viktigt.

Det finns olika kategoriseringar och taxonomier för formella metoder. Vi har valt att utgå från den uppställning baserad på funktionalitet som redovisas i [4, Kap. 2]. Där används fyra kategorier:

- beskrivning och analys
- bevismekanismer
- koppling mellan modell och program
- skalbarhet.

Kategoriseringsmodellen valdes för dess koppling till användning. När Försvarsmakten har behov av att använda formella metoder underlättar kategoriseringen förhoppningsvis valet av metod.

2.1 Metoder för beskrivning och analys

När en specifikation av ett nytt IT-system tas fram kan användningen av formella metoder ge vissa fördelar, som överväger det extra arbete det innebär. De främsta fördelarna är [4]:

- Arbetet med att formalisera specifikationen medför att många fel upptäcks tidigt, vilket sparar tid och därmed pengar.
- Formaliseringen ger en prototyp som kan påvisa fel och brister i specifikationen.
- Prototypen kan provköras genom animering, manipuleras matematiskt eller till och med exekveras, beroende på vilka utvecklingsverktyg som använts. På så sätt går det att påvisa ytterligare fel och brister, utan att någon del i specifikationen behöver implementeras.

De olika processer som kan användas för att utforma en formell specifikation delas in i två grupper utifrån var tyngdpunkten i arbetet ligger. Fokus kan antingen ligga på systemets operationer och hur dessa modifierar det interna tillståndet hos det modellerade systemet, eller så ligger fokus istället på hur data *manipuleras*. Det första fallet där systemets operationer på modellen studeras kallas för *modellbaserad* specifikation. Det andra fallet där datamanipulation står i fokus kallas för *axiomatisk* eller *algebraisk* specifikation.

Deklarativ modellering tillhör kategorin modellbaserad specifikation, men presenteras i ett eget avsnitt i rapporten. Till skillnad från en traditionell modellbaserad specifikationsprocess där modellen byggs upp av en ordnad sekvens av instruktioner, så byggs den deklarativa modellen upp på hög nivå av fakta om modellen. Dessa fakta behöver inte vara ordnade och utifrån dem går det sedan att bygga en fungerande modell för exekvering i dator.

2.1.1 Modellbaserad specifikation

Modellbaserad specifikation fokuserar på det interna tillståndet i det modellerade systemet och hur de operationer som utförs påverkar detta tillstånd. Den matematiska grunden utgörs av; *diskret matematik, mängdlära, kategorilära* och *logik* [4].

Almeida listar fyra huvudgrupper [4, sid. 18] av formella, modellbaserade specifikationer. De är:

Abstrakt tillståndsmaskin (Abstract State Machine (ASM)) som även kallas framväxande algebra (*evolving algebra*). ASM har samma beräkningskraft som en Turingmaskin[6]. En ASM beskrivs av sina interna tillstånd och en ändlig mängd tillståndsövergångar som inte behöver vara deterministiska.

Mängd- och kategorilära beskriver de interna lägena med hjälp av matematiska strukturer såsom mängder, relationer och funktioner. Z och Vienna Development Method (VDM) är två formella metoder som bygger på användningen av mängdlära.

Automatabaserad modellering baseras på det momentana uppförandet hos systemet genom definitioner av hur systemet reagerar på stimuli. Denna typ av modellering är lämplig för system där interna lägen och övergångar är lätt beskrivna.

Modelleringsspråk för realtidssystem är en utökning av automatabaserad modellering och måste kunna hantera ett eller flera fysiska koncept genom sensordata (till exempel tid, kraft och längd). Lustre är ett språk som används för realtidsmodellering och Safety-Critical Application Development Environment (SCADE) är en komplett modelleringsmiljö som bygger på Lustre.

De modellbaserade metoderna bedöms vara de mest använda och mogna av metoderna för beskrivning och analys. Detta är dock en uppskattning baserad på det underlag som erhållits vid de sökningar som gjorts under arbetet med rapporten. I de flesta av de implementationer som studerats har någon form av modellbaserad metod använts.

2.1.2 Deklarativ modellering

Den deklarativa modelleringen kan delas in i tre undergrupper av specifikationsspråk, nämligen logikbaserade språk, funktionella språk och omskrivande språk

[6] En Turingmaskin är en abstraktion av en generell dator och kan utföra alla beräkningsprocesser som en dator kan, även om detta ännu inte är strikt matematiskt bevisat. Turingmaskiner används inom datalogin för att bevisa egenskaper hos någonting, till exempel abstrakta tillståndsmaskiner.

(*rewriting languages*). Liksom övriga formella metoder är de baserade på matematiska grunder.

Logikbaserade språk innehåller ett antal enkla datatyper (till exempel listor) och operationer som beskrivs utifrån sina beteenden på liknande sätt som axiom vid axiomatisk specifikation. Prolog är ett exempel på ett logikbaserat språk.

Funktionella språk bygger på λ-kalkyl[7] och är centrerade kring funktioner. Exempel på funktionella språk är Scheme, Standard MetaLanguage (SML), Haskell och OCaml.

Omskrivande språk är nära besläktade med axiomatiska specifikationsspråk, skillnaden är att axiomen ersatts av ekvationer i funktionsbeskrivningarna. Språken exekverar genom reduktion av ekvationsmängder och liknar i det fallet λ-kalkyl.

2.1.3 Axiomatisk specifikation

Principen för axiomatisk specificering bygger på användning av olika grenar inom multisorterad algebra (*multi-sorted algebras*), vilka består av en samling data som grupperats i mängder. Till detta hör också en uppsättning funktioner som opererar på mängderna, samt en grupp axiom som specificerar de grundläggande egenskaperna hos funktionerna. Grunden till de axiomatiska specifikationerna kommer från matematisk induktion och ekvationslogik (*equational logic*).

Användningen av axiomatiska specifikationer medför att fokus flyttas från de algoritmer som används för att uppnå de önskvärda egenskaperna till representationen av data och hur funktionerna hanterar in- och utdatavärden [4]. Ett exempel på språk som bygger på axiomatiska specifikationsmetoder är Clear [13], som är det första algebraiska specifikationsspråket och har inspirerat till ett otal varianter. Språket OBJ3 [39] tillhör en familj av liknande språk och är i praktiken en implementation av Clear. Språken i OBJ-familjen är exekverbara specifikationsspråk, så kallade ultrahögnivåspråk. Common Algebraic Specification Language (CASL) [22] är ännu ett specifikationsspråk och speciellt inriktat på design av vanlig programvara. CASL är tänkt att fungera som en grund för andra språk och att användas för att skapa en familj av liknande språk. Även Larch [40] utgör en familj av specifikationsspråk. Varje specifikation gjord med Larch innehåller två delar skrivna i olika språk. Ena delen är skriven i Larch gränssnittsspråk och är specialanpassad för ett specifikt programspråk. Den andra delen är skriven i Larch Shared Language (LSL) och utgör en generell del gemensam för alla programspråk. ACT [19, 20] slutligen är även det en språkfamilj som är tänkt att underlätta formell programutveckling. Det ursprungliga ACT ONE har en

[7] Nationalencyklopedin [75] beskriver λ-kalkyl som ett ”formellt logiskt system som bygger på en notation för funktioner föreslagen av Alonzo Church ca 1930”.

utökning som heter ACT TWO som bygger vidare på det första språkets komponenter och gör det mer praktiskt användbart då ACT ONE bara innehåller enkla operatorer vilket begränsar komplexiteten i de specifikationer som kan skrivas.

Det finns en utökning av det axiomatiska ramverket för specifikation, kallad Language Of Temporal Ordering Specification (LOTOS), som möjliggör hantering av samverkande system [4, sid. 24]. Utökningen utgör numera den internationella standarden ISO-8807:1989 [46]. Det finns även en vidareutveckling av LOTOS, kallad Enhanced LOTOS (E-LOTOS), som även den har blivit standardiserad som ISO-15437:2001 [47]. Institut national de recherche en informatique et en automatique (INRIA) i Frankrike har sedan förenklat, förändrat och byggt vidare på LOTOS och skapat LOTOS New Technology i två varianter [17, 91].

2.2 Bevismekanismer

Att bara specificera och konstruera modeller räcker inte för att skapa pålitlighet, det behövs även bevis för korrekt funktionalitet. Detta kallas för *formell verifikation* inom det formella metodområdet. Rushby gör en kategorisering [87, sid. 26] av olika metoder för formell verifikation, som kan sammanfattas på följande sätt:

Nivå 1: Manuella metoder för verifikation, där godkänt resultat uppnås när de som utför granskningen är nöjda med bevisens uppskattade hållbarhet. Naturligt språk kan användas.

Nivå 2: Metoderna baseras på något slags formellt ramverk, men arbetet utförs manuellt. Naturligt språk är inte tillåtet.

Nivå 3: Datorbaserat stöd för både bevis och demonstration av funktionalitet. Det finns formella utvecklingsverktyg för att både uttrycka och föra resonemang kring modeller.

2.2.1 Logik

De tre nivåer av formell verifikation som presenteras av Rushby använder logik på olika sätt. Det finns två huvudinriktningar inom logiken; klassisk logik och intuitionistisk logik.

2.2.1.1 Klassisk logik

Den klassiska logiken bygger på att en utsaga antingen är sann eller falsk. Om ett påstående x används i utsagan[8] $x \vee \neg x$ uppstår en *tautologi*, det vill säga att utsagan är alltid sann. Detta fenomen brukar inom klassisk logik kallas för *lagen om det uteslutna tredje* för att poängtera att det inte finns något tredje alternativ. Lite förenklat kan regeln beskrivas som att antingen saknas en egenskap helt,

[8] Utsagan på naturligt språk lyder x *eller inte* x, det vill säga ”antingen är x sant eller om inte så tar vi motsatsen”, vilket ger värdet sant oavsett vad x har för värde.

eller så finns den. Något tredje alternativ finns inte, egenskapen kan inte både finnas och inte finnas samtidigt.

Satslogiken hanterar enkla satser och huruvida de är sanna eller falska. Satsernas innehåll är egentligen irrelevant. De är uppbyggda av atomära delar och logiken är enkel, så dess användningsområde är begränsat. Logiken byggs upp av premisser (som nödvändigtvis inte behöver vara sanna) som leder fram till slutsatser. Till exempel ger premisserna ”Om solen skiner, så är det varmt ute” och ”Solen skiner” leder fram till slutsatsen ”Det är varmt ute”. Satslogiken används främst för att visa hur ett uttryck påverkar andra.

Första ordningens logik är mer avancerad än satslogiken och innehåller *termer* och *formler*. Dessa kan sättas samman till komplexa uttryck och tillåter kvantifiering av individer, såsom ”det finns” och ”för alla”. I och med möjligheten att kvantifiera, kan uttryck vara sanna eller falska beroende på individuella värden, till exempel $x > x - 1$.

Andra ordningens logik utvidgar första ordningens, hanterar mängder och kan använda variabler för att uttrycka mer komplexa förhållanden. Till exempel hanterar första ordningens logik bara existens av något, medan andra ordningens logik hanterar förhållanden som ”ingår i” och ”är ett element av”.

Logik av högre ordning kan uttrycka ännu mer komplexa satser än första och andra ordningens logik. Det går till exempel utan problem att formulera uttryck som ”varje egenskap som gäller för x också gäller för y”. Tack vare det närmar sig högre ordningens logik programspråk i möjlighet att uttrycka komplexa satser. Användarvänligheten är dock sämre än för riktiga programspråk.

2.2.1.2 Temporallogik

Temporallogik utökar klassisk logik med ytterligare en dimension i form av tid. Det gör att det går att resonera i formella termer om utsagor som ”det kommer att regna”, ”det regnar” och ”det har regnat”. Sanningsvärdet för en formel är därmed inte statiskt och en utsagan är bara sann under tiden för en viss händelse. En typisk temporallogikutsaga är ”idag regnar det, men i morgon regnar det inte”.

Det finns två olika sorters temporallogik; linjär och förgrenad-tids-temporallogik (*computational tree*). Linjär temporallogik hanterar tid som en sekvens av händelser längs en tidslinje. Förgrenad-tids-temporallogik hanterar i stället tiden som en trädstruktur där aktuell tid är rot och varje förgrening motsvarar möjliga alternativa framtida utvecklingar.

2.2.1.3 Intuitionistisk logik

Ett alternativ till klassisk logik är *intuitionistisk* logik, vilken bygger på bevis av utsagor. Alla bevis för utsagor måste dock konstrueras och det går därför inte att veta om utsagan $x \vee \neg x$ är sann eller inte. Det gör att lagen om det uteslutna

tredje[9] inte gäller. Bevis inom intuitionistisk logik liknar på många sätt algoritmer och är därför populära bland forskare inom datavetenskap. Curry-Howardisomorfin[10] kopplar samman intuitionistisk logik och typad λ-kalkyl.

2.2.2 Utvecklingsverktyg för logisk validering

Det finns två motpoler inom logiken som måste balanseras när automatiska utvecklingsverktyg för logisk validering konstrueras: uttryckmöjlighet och enkelhet. Ju större möjligheter ett utvecklingsverktyg har till att uttrycka olika skeenden, desto bättre eftersom mer komplexa skeenden och program då kan uttryckas. Det försvårar dock automatisering på grund av ökande komplexitet. Enkelhet hos vald logik underlättar i stället automatisering, men klarar å andra sidan inte lika komplexa uttryck.

Några programspråk och utvecklingsverktyg som fokuserar på automatiserad deduktion[11] är Prolog [11], ELAN [28, 53] och A Computational Logic for Applicative Common Lisp (ACL2) [51]. Bland dessa utvecklingsverktyg ingår också en klass av Satisfiability Modulo Theories (SMT)-lösare [28, 29] med medlemmar som Simplify-familjen [31] och Yices [98], samt utvecklingsverktygen från Cooperating Validity Checker (CVC)-familjen (till exempel CVC3 [26] och CVC4 [27]). Till Simplify-familjen hör även utvecklingsverktygen Z3 [112] och Alt-Ergo [77].

Några av utvecklingsverktygen för logisk validering kan i vissa fall även tillämpa induktiv slutledning och kan hantera situationer som omfattar slutledning kring oändliga mängder, något som modellvaliderande utvecklingsverktyg inte klarar. En annan egenskap som återfinns hos en del logiskt validerande utvecklingsverktyg är att de tillåter att utvecklingsprocessen går direkt från specifikation till implementation (se mer i Avsnitt 2.3.3). Exempel på sådana utvecklingsverktyg är Prolog och ACL2.

Det finns även utvecklingsverktyg som assisterar vid manuellt valideringsarbete. De består ofta av två moduler, en beviskontrollör och ett bevisutvecklingsverktyg. Utvecklingen av bevisen sker interaktivt genom att i förväg framtagna funktioner för bevismanipulation appliceras sekvensiellt på det ursprungliga beviset. Två exempel på valideringsverktyg är Coq [43] och B-metoden [21].

Formell validering med hjälp av valideringsverktyg kan utföras på två sätt. Det första sättet bygger på *objektlogik*, vilket innebär att användaren själv får definiera den logik som ska användas. Objektlogik är grunden för utvecklingsverktyget Isabelle [2, 45, 80] som använder högre ordningens logik för slutledning. Det

[9] En utsaga kan inte vara sann och falsk samtidigt. Se även Avsnitt 2.2.1.1.

[10] Curry-Howardisomorfi är ett bevis för det direkta sambandet mellan datorprogram och matematisk bevisföring, närmare bestämt typteori och strukturell logik.

[11] Deduktion innebär logisk bevisföring där slutsatser om specifika fall dras ur generella (logiska) regler och används företrädelsevis för matematiska bevis. Motsatsen till deduktion är induktion där fakta (känd genom till exempel observationer) generaliseras till allmänna slutsatser och används framför allt inom experimentella vetenskaper.

andra sättet är att tillhandahålla ett grundläggande språk med tillräckligt hög komplexitet för att det ska gå att uttrycka de flesta matematiska funktioner som krävs. Detta sätt används till exempel i Coq [43].

2.2.3 Modellvalidering

Modellvalidering är en av de vanligare metoderna inom området formella metoder. Den bygger på verifikation av finita automater och använder sig av temporallogik. När verifiering utförs traverseras hela modellen. Antalet tillstånd växer snabbt om modellen utökas vilket gör att metoden inte skalar väl. För att motverka sådana tillståndsexplosioner används olika tekniker. En av dem är *abstraktion*, vilket kommer att behandlas mer i Avsnitt 2.4.

2.2.4 Programlogik och -kommentering

Programkommentarer i den formella metodvärlden är logiska formuleringar av kraven som bäddas in i koden. Principen bygger på *Hoarelogik*[12] och basen för kommentarerna är generell. Varje programspråk har en egen variant av logiken, det vill säga semantiken är programspråksspecifik. De inbäddade kommentarerna ska visa de villkor som ska vara uppfyllda innan kodavsnittet ska exekveras och det logiska tillstånd som programmet ska vara i efter exekvering av kodavsnittet.

Fördelen med programkommenteringsparadigmet är att källkoden innehåller grunden för valideringen, i stället för att den utgörs av en separat specifikation. Modellen som valideras konstrueras utifrån källkoden och dess kommentarer tillsammans med underliggande specifikation av programspråket. De saker som hör samman följs åt och kan hanteras samtidigt, vilket innebär smidigare hantering och mindre risk för misstag.

Tanken bakom kommentering och Hoarelogik gav upphov till en speciell form av kommentering som kallas *kontrakt*. Det första programspråk som använde sig av kontrakt var Eiffel [34], som tillämpade dynamisk verifikation av kontrakt under exekvering. Konceptet har vuxit och kontrakt används numera i många av de stora programspråken. Tillsammans med speciella utvecklingsverktyg eller varianter används konceptet i till exempel ADA (genom varianten SPARK [38, 95, 96, 97]), C# (genom varianten SPEL# [62]), Java (genom varianterna Esc/Java [36], KeY [52] och Krakatoa [59]) och C (genom utvecklingsverktygen Verifying Concurrent C (VCC) [23, 24, 25] och Frama-C [8], samt språket ACSL [9]). Utvecklingsverktyg för att statiskt kontrollera överensstämmelsen mellan koden och kontraktet kan vara helt bevisbaserade, men det finns även några utvecklingsverktyg som kombinerar modellverifiering och bevisassistans, till exempel Loop [10] och Bandera [33].

[12] Hoarelogik är ett enkelt strukturerat programspråk baserat på formell logik. [73]

2.3 Koppling mellan modell och program

En viktig detalj i utvecklingsprocessen när formella metoder används är att koppla specifikationen till färdig programvara, så att de egenskaper som specificerats verkligen också implementerats på rätt sätt. Det finns två sätt att hantera transformationen från specifikation till programkod på, antingen är specifikationen skriven i ett programspråk och kan köras direkt eller så måste transformationen hanteras separat. I det senare fallet uppstår problemet med en korrekt transformation vilket avhjälps genom användning av formella metoder.

Riktigheten i transformationen kan hanteras genom en noggrant kontrollerad (formellt verifierad) transformationsprocess (*konstruktion*) för att på så sätt implicit kunna garantera riktigheten hos implementationen. Detta kallas på engelska för *correct-by-construction*. Ett annat sätt är att i stället fokusera på implementationen och med hjälp av matematiska bevis skapa krav på implementationen utifrån specifikationen (*verifikation*). Om implementationen kan bevisas uppfylla kraven är allt i sin ordning och riktigheten hos transformationsprocessen blir irrelevant.

2.3.1 Förfining

Förfining (*refinement*) bygger på en stegvis övergång från specifikation till färdig programkod. Varje steg innebär ett val, till exempel av datatyp för en variabel eller vilken algoritm som ska användas. Alla steg måste åtföljas av ett bevis för deras riktighet, vilket ofta görs genom att skapa matematiska beviskrav som implementationen måste uppfylla. Denna teknik används av de formella ramverken Z, VDM och B. Förfining är populärt inom den kommersiella världen och stöds därför av en mängd utvecklingsverktyg.

2.3.2 Extrahering

Konceptet extrahering är baserat på det faktum att Curry-Howardisomorfin visar ett samband mellan λ-kalkyl och högre ordningens logik. Det formella metodramverket kring Coq bygger på *induktiv konstruktionskalkyl*, Calculus of Inductive Construction (CIC) på engelska, vilket är en utökning av λ-kalkylen. Varje logisk sats kan därför också ses som en specifikation och varje bevis som kan uttryckas med hjälp av CIC också är ett program som uppfyller en specifikation. Följaktligen kan den kalkylmässiga delen av bevisen extraheras och överföras till ett programspråk. I Coqs fall kan kalkyldelen extraheras till Scheme, Haskell eller OCaml. Extraheringen kan dessutom göras automatiskt och i ett steg.

2.3.3 Exekvering

Genom att använda deklarativa programspråk som Scheme, ACL2, Haskell, SML och OCaml samt logikbaserade språk som Prolog går det att gå direkt från specifikation till implementation. Specifikationen skrivs med hjälp av något av dessa

språk och resultatet är en de-facto implementering av specifikationen. Konceptet har dock inte rönt någon större framgång eftersom den kommersiella världen sällan betraktar dessa programspråk som praktiska alternativ för de system de vill implementera.

2.4 Skalbarhet

För att underlätta användningen av formella metoder och även till viss del utöka mängden av tillämpningar behöver gjorda specifikationer kunna skalas upp eller ner, det vill säga omvandlas för att lättare kunna hanteras. Detta innebär till exempel att ovidkommande detaljer kan abstraheras bort, eller att detaljgraden kan ökas. Grunden till detta är teorin bakom *abstrakt tolkning (abstract interpretation)*, vilken gör det möjligt att göra korrekta approximeringar.

En vanlig metod för att hantera stora och komplexa problem är att bryta ner dem i delproblem, som sedan löses var för sig. Principen kan även användas inom de formella metoderna. Utvecklingsverktygsutbudet för sådana upp- eller nerskalningar är dock litet, delvis beroende på att de abstrakta tolkningarna lätt blir obestämbara. Dessutom är omvandlingsprocessen tätt knuten till vilka egenskaper som ska bevisas och vilket språk som används för modelleringen.

Ett exempel på utvecklingsverktyg för omvandling är JaKarTa [7], som används för bevisföring kring JavaCard-specifikationer. Det utgår från effekter på aktuell modells datastrukturer vid modelltransformation och erbjuder ett regelbaserat språk och mekanismer för att specificera transformation av ad hocmodeller.

2.5 Semi-formella metoder

De semi-formella metoderna är nära förknippade med formella metoder. Begreppet myntades under sent 1960-tal och var en reaktion på de problem som uppstod i och med att programmen ökade i storlek och komplexitet. Det som behövdes var ett strukturerat sätt att hantera utvecklingsprocessen på, skapa överblick, jämka samman olika beskrivningssätt till en mer enhetlig form och till viss del formalisera begrepp och uttryck till ett gemensamt språk. Exempel på semi-formella metoder är

- UML
- Structured Systems Analysis and Design Method (SSADM)
- Profitable Information by Design (PRIDE)
- Nastec Structured Analysis & Design
- System Development Methodology (SDM)
- Spectrum.

De semi-formella metoderna är strukturerade, men saknar matematisk notation och använder istället naturligt språk och en strukturerad (grafisk) notation. Det gör att semi-formella metoder är lättare än formella att förstå för människor, men att de har problem med tvetydighet. De semi-formella metoderna är därför inte lämpliga att använda när fullständig pålitlighet i utvecklingsprocessen är ett krav.

Det har sedan 1990-talet pågått forskning kring hur semi-formella och formella metoder kan kombineras så att deras respektive fördelar kan utnyttjas [86, 110, 111, 113]. Målet med forskningen är att ta de användarvänliga delarna (främst den grafiska notationen) från de semi-formella metoderna och göra dem entydiga, för att underlätta den viktiga specifikationsdelen av utvecklingsprocessen. Entydigheten bör också innebära att transformationen från specifikation till färdig kod kan automatiseras och utföras av en dator. Detta är en intressant forskningsmässig utmaning som i förlängningen kan leda till ökad användarvänlighet hos de formella metoderna.

3 Standarder

I detta avsnitt presenteras några av de standarder för IT-säkerhet som tagits fram. Vissa av dem, som Common Criteria for Information Technology Security Evaluation (CC), har fått ett stort genomslag. Urvalet är baserat på författarnas uppfattning om vad som är de mer framträdande standarderna efter studier av området.

Två standarder som är värda att omnämnas, men som inte bedömdes tillräckligt intressanta för att ges egna avsnitt, är Institute of Electrical and Electronics Engineers (IEEE) 1028-1997 *IEEE Standard for Software Reviews* som specificerar hur manuell kodgranskning ska gå till och the Orange Book vars officiella namn är *Trusted Computer System Evaluation Criteria (TCSEC)* och ingår i the Rainbow Series med IT-säkerhetsrelaterade standarder utgiven av Department of Defense (DoD) i USA. The Orange Book gavs ut 1983 och ersattes 2005 av CC.

3.1 Common Criteria

Common Criteria for Information Technology Security Evaluation (CC) är en internationell standard (ISO/IEC 15408) för evaluering av säkerhetsegenskaper hos IT-produkter och system.

Inom ramen för en IT-produkts säkerhetsfunktionalitet är målet att skydda tillgångar, vilket kräver kunskap om vilka hot som finns och de risker de leder till. Sedan ska riskerna reduceras genom lämpliga (mot)åtgärder. De åtgärder som identifieras motsvarar säkerhetskraven för IT-produkten. Med CC kontrolleras både att de säkerhetskrav som ställs på en produkt är tillräckliga för den hotbild som finns och att de implementeras korrekt. CC fokuserar på att skydda tillgångarna med avseende på sekretess, integritet och tillgänglighet.

Inom CC finns följande nyckelbegrepp:

Target of Evaluation (TOE) IT-produkten eller systemet under utvärdering.

Security Functional Requirement (SFR) Säkerhetsrelaterade funktionskrav som kan ställas på en produkt. CC tillhandahåller en lista med standardfunktionskrav.

Security Assurance Requirement (SAR) Säkerhetsrelaterade assuranskrav som ligger till grund för utvärdering av korrekthet hos en TOE (med avseende på de säkerhetsmässiga funktionskrav som utvärderas). Exempel på utvärderingsaktiviteter är att testa TOE:n eller att undersöka olika designrepresentationer av TOE:n. En SAR är kopplad till en specifik EAL-nivå.

Det går också att uppnå en högre EAL-nivå genom att lägga mer ansträngning i utvärderingen avseende på omfattning, noggrannhet och djup.

Security Target (ST) Ett dokument som innehåller säkerhetskrav för den TOE som utvärderas. ST är implementationsberoende och inkluderar både SFR och SAR. Den innehåller även hotbild med mera som definierar det aktuella säkerhetsproblem som studeras.

Protection Profile (PP) Ett dokument som innehåller säkerhetskrav för en typ av TOE (till exempel för brandväggar). PP är implementationsoberoende och är vanligtvis skrivet av en användare eller användargrupp. En TOE kan utvärderas mot en eller flera PP. PP används också som mall för ST.

Evaluation Assurance Level (EAL) Omfattning, djup och noggrannhet av en utvärdering ger ett numerisk betyg i form av en EAL. Totalt finns det sju EAL:er och varje EAL motsvarar ett paket av SAR:er, som täcker hela utvecklingen av en produkt, med en viss grad av noggrannhet.

I korthet erbjuder CC ett standardiserat sätt att beskriva en produkts funktionella säkerhetskrav och genom Evaluation Assurance Level (EAL)-nivåerna ge ett mått på hur troligt det är att produkten uppfyller säkerhetskraven. CC erbjuder även ett standardiserat sätt att utvärdera produkten på.

En produktutvecklare sammanställer sina Security Functional Requirement (SFR) och Security Assurance Requirement (SAR) i en Security Target (ST), tillsammans med en beskrivning av Target of Evaluation (TOE):n, operativ miljö och de hot som anses finnas. En användare, användargrupp eller organisation kan göra samma sak för en typ av TOE, vilket resulterar i en Protection Profile (PP). En PP innehåller krav som ska vara uppfyllda i en produkt och beskrivningar av hot som anses relevanta. En PP kan också utvärderas inom ramen av CC för att bestämma om den kan anses komplett, konsistent och tekniskt korrekt. För att göra en produkt attraktiv för användare, kan en produktutvecklare använda en eller flera PP när sammanställningen av ST:n görs (och därigenom när beslut tas om vilka funktionskrav som produkten ska ha och vilka assuranskrav som ska gälla).

En ST utvärderas för att bestämma om de säkerhetskrav som finns är tillräckliga för den hotbild som anges. Hotbilden kan bygga på situationsanpassade hotanalyser, en eller flera PP:n eller en kombination därav. TOE evalueras för att se till att det finns tillräcklig assurans för att SFR:erna i TOE:ns ST är uppfyllda. Assuransnivån ges av en EAL. CC skiljer mellan en TOE och operativ miljö. I ST:n måste både TOE:n och operativa miljöns definieras tillsammans med de krav som ställs för var och en. Dock är det bara TOE:ns krav som utvärderas. Vid utvärderingen antas att de krav som ställs på den operativa miljön tillsammans med de antaganden som gjorts om densamma uppfylls och att allt är korrekt implementerat. Samtidigt kan TOE:n ha ytterligare säkerhetskrav som inte finns i ST:n, och de utvärderas därför heller inte. Som resultat av utvärderingen framkommer

även om en ST överensstämmer med en eller flera PP (dock endast för de PP som finns angivna i ST:n).

För att utvärderingen ska vara så standardiserad som möjligt används Common Evaluation Methodology (CEM), som också är en del av CC. De laboratorier som utvärderar IT-produkter mot CC måste följa standarden ISO 17025. Certifieringen ges av en nationell godkännandemyndighet.

CC inkluderar endast administrativa säkerhetsåtgärder som är direkt relaterade till IT-säkerhetsfunktionalitet och bedömer inte kvaliteten på de kryptografiska algoritmer eller moduler som används. För kryptografiska moduler finns andra standarder, till exempel Federal Information Processing Standards (FIPS) 140-2, som beskrivs närmare i Avsnitt 3.2.

Det finns en rad certifierade operativsystem, till exempel är Apple Mac OS X 10.6 CC certifierat för nivå EAL 3+[1] och Microsoft Windows 7 för nivå EAL 4+. Det är viktigt att komma ihåg att TOE:n bara evalueras mot en rad givna säkerhetsfunktioner och att vissa antaganden om den operativa miljön har gjorts. Säkerhetsuppdateringar är därför nödvändigt även för certifierade operativsystem.

Det har framförts kritik mot CC, bland annat anses det vara en dyr process. Den anses även vara så tidskrävande att när produkten till slut blivit certifierad så har den hunnit bli föråldrad, med lägre säkerhet som resultat. Det finns också viss kritik mot att utvärderingen mest fokuserar på produktens dokumentation och inte på själva produkten, speciellt för de lägsta assuransnivåerna, EAL 1–4 [48]. Även hela konceptet med certifiering har ifrågasatts [68] i och med att det påvisats brister i redan certifierade system.

I Storbritannien finns några alternativ till CC som ska vara snabbare och billigare att använda. Till exempel har Communications-Electronics Security Group (CESG) tagit fram CESG Tailored Assurance Service (CTAS) [16] och CESG Commercial Product Assurance (CCPA) [15].

3.2 Standarden FIPS 140-2

FIPS 140-2 ”Security Requirements for Cryptographic Modules” är den amerikanska regeringens datasäkerhetstandard som används för att ackreditera kryptografiska moduler. Denna standard specificerar de säkerhetskrav som ska uppfyllas av en kryptografisk modul som används inom ett säkerhetssystem som skyddar känslig, men oklassificerad data. Standarden inkluderar fyra kvalitativa säkerhetsnivåer (1–4, där 4 är högst). Nivå 1 ställer endast krav på användning av minst en godkänd algoritm eller funktion, samt användning av produktionsfärdig hårdvara. På nivå 4 krävs att de fysiska säkerhetsmekanismerna ger ett komplett skydd för den kryptografiska modulen med avsikt att upptäcka och reagera på alla obehöriga försök till fysiskt intrång.

[1] Ett ”+” betyder att den uppfyller ytterligare SAR:er utöver de som specificeras i EAL:en.

FIPS 140-2 tillhandahåller listor med godkända algoritmer, säkerhetsfunktioner, slumpgeneratorer och tekniker för nyckelöverenskommelse.

För att validera kryptografiska moduler mot FIPS 140-2 eller andra standarder används Cryptographic Module Validation Program (CMVP). Kryptografiska moduler valideras i ackrediterade laboratorier. En svaghet hos standarden som även finns för de höga säkerhetsnivåerna är att den kryptografiska modulen kan ha sidokanalssårbarheter som möjliggör enkel extraktion av nycklar, till exempel genom *differential power analysis*.

En ny version av standarden, FIPS 140-3, är under utveckling och ska så småningom ersätta FIPS 140-2. FIPS 140-3 tillför en extra säkerhetsnivå och nya säkerhetsfunktioner. Den höjer också kraven på försvar mot sidokanalssårbarheter och har ett separat kapitel om programvarusäkerhet.

3.3 DO-178C/ED-12C

Flygindustrin har en egen standard för utveckling av kritisk programvara. Standarden heter DO-178C/ED-12C och fastställdes i december 2011. Den liknar till stora delar sin föregångare, DO-178B, men har uppdaterats och bearbetats för att korrigera brister i den tidigare versionen. Bland annat har den blivit tydligare och en del tvetydigheter har arbetats bort. Den har också utökats genom tillägg av bilagor och tekniska komplement och omfattar nu över 600 sidor. Genom att använda tekniska komplement kan standarden hållas aktuell utifrån tekniska landvinningar. Varje tekniskt komplement beskriver en specifik teknik och dess tillämpning inom standarden. För närvarande finns följande bilagor och tekniska komplement som är relevanta för programutveckling:

DO-330 *Software Tool Qualification Considerations*, vilken behandlar kriterier för hur användningen av utvecklingsverktyg ur alla aspekter (urval, testning, utveckling m.m.) ska gå till. I DO-178C/ED-12C finns sedan kriterier för när DO-330 är tillämplig.

DO-331 *Model-Based Development and Verification Supplement to DO-178C and DO-278*, vilken behandlar modellbaserad utveckling och verifiering som verktyg för att undvika vanliga utvecklingsmässiga fallgropar.

DO-332 *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*, vilken behandlar objektorienterade programvaror och under vilka villkor de kan användas för utveckling.

DO-333 *Formal Methods Supplement to DO-178C and DO-278A*, vilken behandlar villkor och krav för användning av formella metoder under (delar av) utvecklingsprocessen.

Den tidigare standarden DO-178B fick kritik för att vara oklar och sakna tillräckligt djup kring nyckelbegrepp som härledda krav och hög- och lågnivåkrav. Även

gränsytorna mellan systemkrav och systemdesign, liksom programvarukrav och programvarudesign var oklara. Dessa brister ska vara åtgärdade i DO-178C [84].

3.4 Administrativa standarder

De standarder som beskrivs i Avsnitt 3.1–3.3 är främst inriktade på den tekniska delen av IT-säkerhet. Det finns även standarder som syftar till att hantera säkerheten kring IT-systemen genom förbättrade administrativa rutiner. Några av dessa beskrivs nedan.

3.4.1 ISO/IEC 27000-serien

ISO/IEC 27000-serien för informationssäkerhetsstandarder ger rekommendationer för bästa praxis om hantering, risker och kontroller för informationssäkerhet inom ramen för ett Ledningssystem för informationssäkerhet (LIS). Den svenska motsvarigheten till ISO/IEC 27000-serien heter SS-ISO/IEC 27000.

3.4.2 BITS – Basnivå för informationssäkerhet

Basnivå för informationssäkerhet (BITS) [54] är rekommendationer från Myndigheten för samhällsskydd och beredskap (MSB) på ett antal administrativa säkerhetsåtgärder som en organisation bör genomföra för att uppnå en acceptabel säkerhetsnivå för hantering av information inom organisationen. BITS utgör alltså en lägstanivå att bygga vidare från.

Det finns också ett analysverktyg, BITS Plus, för informationssäkerhet som utifrån kraven på sekretess, riktighet och tillgänglighet genererar åtgärdsförslag som svarar mot vald kravnivå.

3.4.3 ISO/IEC 13335

ISO/IEC 13335 ”IT-säkerhetstekniker – styrning och kontroll av informations- och kommunikationsteknikssäkerhet” presenterar modeller och begrepp för en grundläggande förståelse av informations- och kommunikationsteknikssäkerhet. Den tar upp generella ledningsfrågor som är viktiga för en framgångsrik planering, genomförande och drift av IT-säkerhet. Vissa delar av standarden har ersatts av standarder i ISO/IEC 27000-serien.

4 Organisationer, initiativ och projekt för ökad pålitlighet

Det finns ett flertal grupper som stödjer utvecklingen av IT-säkerhetsarbetet i samhälle och industri. Några av de mer kända initiativen presenteras i detta avsnitt. Urvalet är baserat på författarnas egna uppfattningar om de i litteraturen vanligast förekommande grupperna och initiativen efter de studier som genomförts inom projektet.

Flera av de presenterade organisationerna, initiativen och projekten använder termen programvaruassurans (*software assurance*) i sina namn. I korthet innebär programvaruassurans användning av tekniker och metoder för att säkerställa att programvaran fungerar som den ska utan att innehålla sårbarheter, skadlig kod, eller defekter. Under begreppet samlas både själva egenskapen av assurans och processen som behövs för programvaruassurans. Assurans definieras som:

> *Justifiable confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle and that the software functions in the intended manner.* [71]

Processen som behövs för programvaruassurans definieras som:

> *[. . .] the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures.* [70]

Programvaruassurans motsvarar väsentligen samma saker som vi i rapporten lagt i begreppet pålitlighet.

4.1 Trusted Computing Group

I oktober 1999 bildades Trusted Computing Platform Alliance (TCPA) och mindre än två år senare släpptes deras första specifikation av Trusted Platform Module (TPM) [66]. 2003 ersattes TCPA av Trusted Computing Group (TCG) som till att börja med hade 14 medlemmar [85]. TCG tog då över TPM-specifikationen och vidareutvecklar den. TCG:s mål är att utveckla teknik och metoder för att upprätthålla tilltron till att programvara och system är säkra att använda. Mekanismerna är kraftfulla och måste kontrolleras av plattformsägarna. Kraven från användarna är mycket olika och ger en komplex miljö för TPM att verka i.

TPM består oftast av en integrerad krets på moderkortet i en dator. Kretsen fungerar tillsammans med Basic Input/Output System (BIOS) som rot för alla pålitlighetsfunktioner i datorn. Kretsen har skydd mot manipulering och används av programvara som erbjuder pålitlighetsfunktionalitet. TPM är konstruerad så att alla säkerhetskritiska delar av processen alltid sker i kretsen, inklusive kryptering, dekryptering, hantering av nycklar och upprätthållande av diverse säkerhetspolicyer, som en garanti mot manipulation. Om pålitlighetsmekanismerna stängs av skyddas fortfarande innehållet i kretsen mot otillåten läsning.

TCG har vidareutvecklat TPM-konceptet [18] och bland annat tagit hänsyn till kritik som rör svagheter i skyddet av den personliga integriteten för användare av utrustning med TPM-chip. Den nya teknik som TCG tagit fram kallas Direct Anonymous Attestation (DAA). TCG:s skyddsmekanismer måste tillåta igenkänning av en specifik plattform, men samtidigt skydda mot identifiering av densamma för att undvika att personlig information läcker eller att ett loggspår kan kopplas till en fysisk person.

4.2 Trustworthy Software Initiative

Trustworthy Software Initiative (TSI) är ”UK's Public Good activity for Making Software Better” [102]. De skriver att deras mål är att

> *enhance the overall software and systems culture, with the objective that software should be designed, implemented and maintained in a trustworthy manner.* [65]

Bland TSI:s bidrag till ökad pålitlighet i programvara finns ett ramverk för pålitlig programvara (*Trustworthy Software Framework*), en standard för hur pålitlig programvara ska byggas (PAS754:2014 ”Software Trustworthiness – Governance and management – Specification” [105]), samt en kunskapsdatabas (*Trustworthy Software Knowledge-base*).

4.3 Department of Homeland Security Software Assurance Program

USA:s Department of Homeland Security (DHS), har ett program för programvaruassurans kallad Software Assurance Program (SAP) [32]. Programmet försöker, på samma sätt som i fallet med Software Assurance Forum for Excellence in Code (SAFECode), reducera antalet sårbarheter, minimera sårbarheternas exploatering och underlätta utvecklingen och driftsättning av pålitlig programvara. Ett viktig mål är att ändra dagens säkerhetstänkande från patchhantering till programvaruassurans.

Programvaruassurans inkluderar följande komponenter [32]:

Individer: Utbildning för både utvecklare och användare.

Processer: Praktiska riktlinjer och bästa praxis för utveckling av säker programvara.

Teknik: Utvecklingsverktyg för utvärdering av sårbarheter i programvara och programvarukvalitet.

Inköp: Specifikationer och riktlinjer för inköp och outsourcing.

SAP:s fokus ligger på att tillhandahålla dokumentation om bästa praxis, men de har också en rad utvecklingsverktyg för modellering, kodanalys och tester enligt principen om svarta lådor.

Build Security In [12] är en av programmets projekt som fokuserar på att undvika de 25 säkerhetsmässigt allvarligaste programvarufelen [1]. Det finns även ett privat initiativ med liknande inriktning och namn, Building Security In Maturity Model (BSIMM) [60]. De har för närvarande (2015-12-18) 112 säkerhetsrelaterade punkter att ta hänsyn till i utvecklingsprocessen. På sin hemsida skriver de att ”BSIMM is designed to help you understand, measure, and plan a software security initiative”. Som underlag har de använt data från de ledande programsäkerhetsinitiativen i världen, sammanställt den och på så sätt fått fram sina rekommendationer.

4.4 National Institute of Standards and Technology och Software Assurance Metrics And Tool Evaluation

DHS National Cyber Security Division och National Institute of Standards and Technology (NIST) sponsrar Software Assurance Metrics And Tool Evaluation (SAMATE)-projektet [94].

De två målen med SAMATE är:

- Utveckling av en måttenhet för mätning av effektiviteten hos utvecklingsverktyg för programvarusäkerhetsbedömning, Software Security Assessment (SSA).
- Bedöma nuvarande SSA-metoder och utvecklingsverktyg för att identifiera brister som kan leda till sårbarheter och fel i programvaruprodukter.

SAMATE har jobbat med att utveckla en metod för utvärdering av utvecklingsverktyg för programvaruassurans. De gör det genom att utveckla verktygsspecifikationer, testplaner och testuppsättningar. Olika verktygsutvecklare kan därför testa och utvärdera sina utvecklingsverktyg. Utvärderingarna blir sedan tillgängliga genom SAMATE så att programutvecklare lättare kan välja rätt utvecklingsverktyg. SAMATE organiserar också en workshop, Static Analysis Tool Exposition, där resultat presenteras.

Projektet tillhandahåller också en referensdatabas med kända säkerhetsbrister och tillhörande fixar så att utvecklare kan testa sina metoder och användare kan utvärdera utvecklingsverktyg. Databasen innehåller över 80 000 testfall.

4.5 Software Assurance Forum for Excellence in Code

Software Assurance Forum for Excellence in Code (SAFECode) är en ideell organisation med mål att öka förtroendet i informations- och kommunikationsprodukter genom att förespråka beprövade assuranstekniker för programutveckling. Organisationen har publicerat en rad artiklar om programvaruassurans och håller kurser i säker programmering.

5 Programspråk

Det går att komma ganska långt med att ha ett säkerhetstänkande vid programmering. Dessutom har det utvecklats standarder och riktlinjer för säker programmering som ytterligare förbättrar sannolikheten för ett lyckat resultat. Utöver dessa finns speciellt utvecklade programspråk med inbyggda säkerhetsfunktioner och även vissa språk som är designade för specifika säkerhetsbehov. Detta kapitel presenterar några olika typer av sådana programspråk.

5.1 Säkerhetstypade programspråk

Typning används vid programmering och är en egenskap hos programspråken. Egenskapen är kopplad till de krav som språket ställer när ny data (variabler, objekt med mera) skapas. Ett typat programspråk kräver att en variabel knyts till en specifik typ (kategori av data). Exempel på sådana typer är heltal, flyttal och strängar. Det finns även ett fåtal språk som saknar typning. Typning används för att höja kvaliteten på programkod genom att förhindra operationer på inkompatibla datatyper.

Säkerhetstypade programspråk (*security-typed programming languages*) [116] använder statisk typning för att upprätthålla säkerhetsriktlinjerna för informationsflödet. Dessa språk tillåter programmeraren att ange sekretess- och integritetsbegränsningar för information som används i ett program och kompilatorn verifierar att programmet uppfyller begränsningarna.

Den vanligaste säkerhetsmekanismen för informationssekretess är åtkomstkontroll där vissa rättigheter behövs för att få tillgång till information. Åtkomstkontrollmekanismer begränsar åtkomsten till information men begränsar inte informationsflödet efter att tillgång till informationen har beviljats. Till exempel kan en datafil först läsas och sen kopieras till ospecificerad plats efter att användaren har fått tillgång till den. Åtkomstkontrollmekanismer kan därför inte skydda informationssekretessen under informationens hela livslängd. För att uppnå detta måste informationens flöde inom programmet kontrolleras, vilket kallas för *säkert informationsflöde*[1]. För mer ingående information om språkbaserad informationsflödeskontroll rekommenderas översiktsartiklarna av Hedin [41] och Sabelfeld och Myers [88].

Informationen kan flöda i ett program genom implicita och explicita flöden. Explicita flöden innebär att information kopieras från en variabel till en annan. Ett exempel på ett explicit flöde är `y = x + 2`. Om x skulle vara en hemlig och y en publik variabel, skulle information om x läcka genom y.

[1] FOI-rapporten ”Metoder för informations- och åtkomstkontroll”, FOI-R--4010--SE, ger mer information om säkra informationsflöden.

Ett implicit flöde uppstår när kontrollflödet i programmet påverkas av hemliga värden. Det vill säga att det hemliga x inte direkt ingår i beräkningen av det publika y, men ändå ingår genom en indirekt påverkan. Ett typiskt exempel är en villkorssats där värdet på y förändras beroende på värdet på x, till exempel:

```
y = false
if x then
   y = true
print y
```

På grund av det implicita flödet går det att ta reda på information om x genom att titta på y eftersom y indirekt påverkas av x.

Riktlinjer som säger att publik utdata inte får påverkas av hemlig indata kallas riktlinjer för *non-interference*. Men hjälp av sådana riktlinjer kan program behandla hemlig information så länge publika utdata inte avslöjar information om de hemliga värdena. Non-interference är målet inom säkert informationsflöde.

Säkerhetstypade programspråk använder statisk informationsflödeskontroll [30] och typsystem [82]. Säkerhetstypning innebär att varje uttryck i ett program har en säkerhetstyp som inkluderar en vanlig typ så som int eller boolean och en etikett (*label*) som specificerar riktlinjer om hur värdet kan användas.

Sekretesshanteringen hos säkerhetstypade programspråk kontrolleras genom en statisk typkontroll där kompilatorn använder säkerhetstypningen för att verifiera att programmet inte innehåller felaktiga informationsflöden. För att även kunna kontrollera implicita flöden används en etikett som håller reda på aktuella beroenden mellan variabler och var i koden de inträffar med hjälp av programräknaren. I det tidigare exemplet på ett implicit flöde skulle programräknaretiketten sättas med avseende på x:s etikett och då skulle alla uttryck inom villkorssatsen (if-satsen) behöva ha minst samma sekretessnivå som x för att kunna anses som säkra.

Idén bakom de säkerhetstypade programspråken bygger på konceptet att tilltro till ett programs sekretessfunktionalitet borde komma från en rigorös analys som visar att det kan upprätthålla de satta sekretessriktlinjerna. Det finns redan utvecklingsverktyg baserade på olika formella metoder som kan bevisa att säkerhetstypade system kan upprätthålla non-interference. Dessa formella metoder kallas funktionella programspråk [4, 74], vars kärna utgörs av λ-kalkyl. Scheme, SML, Haskell och OCaml är exempel på sådana språk. Några exempel på utvecklingsverktyg för hantering av säkerhetstypning är:

Jif En utökning av programspråket Java som ger stöd för informationsflödeskontroll och åtkomstkontroll, upprätthållen både vid kompilering och körning.

Jämfört med Flow Caml och SPARK/Ada har Jif [50] ett utökat stöd för att koppla säkerhetsrelaterade krav från specifikationen till färdiga program. Jif har monomorfa[2] typer och utför endast lokal typrekonstruktion[3] (*type reconstruction*) [92].

Flow Caml En informationsflödesanalysator för programspråket Caml [99]. Flow Caml har polymorfa[4] typer och har en full typhärledningsalgoritm[5], till skillnad från Jif [37].

SPARK En uppsättning utvecklingsverktyg med mera för programutveckling och en delmängd av programspråket Ada. Typningen i SPARK är ännu striktare än den redan starka typningen i Ada. Det går därför att använda SPARK som ett analysverktyg för informationsflöden [38, 95, 96, 97].

Vissa sekretessriktlinjer kan vara dynamiska och okända vid kompilering, till exempel har filer i filsystemet rättighetsflaggor som kan ändras över tid. För att hantera detta har användning av dynamiska säkerhetsriktlinjer föreslagits [69]. Det medför att de etiketter som håller ordning på beroenden kan vara variabler i sig. Likaså blir de säkerhetstyper som är beroende av värden som beräknas vid körning omvandlade till beroendetyper som kontrolleras vid körning.

En annan egenskap som är användbar är nedgradering av sekretessnivån för specifik information. Utan möjlighet till nedgradering i vissa situationer blir systemet i praktiken oanvändbart. Nedgradering behövs till exempel vid beräkning och publicering av statistik över hemliga data. Det kan också handla om att användare ska kunna logga in i systemet. Det faktum att en användare får logga in läcker information om lösenordet [89]. I lösenordsfallet kan argumentationen verka väl teoretisk, tanken är att läckaget sker genom att användaren får kännedom om att lösenordet är korrekt. Det är trots det en viktig sak att belysa för att visa på behovet av robust nedgradering för att motverka onödigt informationsläckage, men fortfarande tillåta nödvändigt.

Det dyker ständigt upp nya utvecklingsverktyg för säker programmering. Det är dock inte alltid som de överlever någon längre tid. Detta ökar risken för att personalen ständigt behöver lära om. Bristande kontinuitet är också en felkälla, något som inte gynnar pålitligheten hos programvaran. De ovan beskrivna verktygen för säkerhetstypning har alla funnits under en relativt lång tid och underhålls för närvarande aktivt.

[2]Monomorfa betyder enkelformiga, oföränderliga.

[3]Inom datavetenskapen används termen *lokal* för att indikera att variabler och typer bara är giltiga inom den specifika funktion där de definieras. Innebörden av ett variabelnamn eller en typ kan alltså skilja sig mellan de funktioner där de används. Typrekonstruktion innebär att kompilatorn automatiskt härleder en variabels typ.

[4]Polymorfa betyder mångformiga, föränderliga.

[5]Full typhärledning innebär att kompilatorn automatiskt härleder typen hos både lokal och globala variabler om typen inte är specificerad.

5.2 Andra säkra programspråk

Principen för säkra informationsflöden ligger även till grund för en grupp programspråk och ramverk inriktade mot säkra distribuerade system. Flera av dessa bygger på Jif och är således säkerhetstypade programspråk i grunden. En viktig egenskap är möjligheten att kunna köra tredjepartskod (även kallad mobil kod) på ett säkert sätt.

Några av dessa språk och ramverk är:

E är ett programspråk för säker distribuerad databehandling. E bygger på en ren objektmodell och använder en förmågebaserad (*capability-based*) säkerhet som kallas objekt-förmågor. Upphovsmännen bakom E beskriver objektförmågorna som en striktare tillämpad form av objektorienterad programmering. De skriver också att E:s arkitektur gör att det inte kan uppstå ömsesidig låsning (*deadlock*) av resurser, att distribuerade parter som inte litar på varandra ändå kan samverka på ett säkert sätt och att det går att köra tredjepartskod på ett säkert sätt i ett system [108, 109].

Fabric är ett programspråk som utökar Jif för att bygga säkra distribuerade system. Enligt skaparna av Fabric kan det ge säkerhetsgarantier till systemanvändare som misstror varandra. Dessa säkerhetsgarantier är giltiga även i ett decentraliserat system som saknar centraliserad säkerhetsverkställighetsmekanism (*security enforcement mechanism*). Genom ett kontrollerat informationsflöde kan även tredjepartskod köras på ett säkert sätt [100].

Swift baseras på Jif och är ett ramverk för att bygga säkra webbtillämpningar. Swift delar automatiskt webbtillämpningen på ett säkert sätt mellan klienten och webbservern. Kodplaceringen begränsas av deklarativa informationsflödesriktlinjer på en hög nivå som strikt upprätthåller sekretessen och integriteten av data på serversidan. När Swift delar upp koden optimerar den körbarhet och effektivitet under bivillkoret att säkerheten bibehålls [101].

Liksom för utvecklingsverktygen för säkerhetstypning skapas det kontinuerligt nya programspråk som sägs vara säkrare och enklare. Även valet av programspråk bör styras av faktorer såsom allmänt genomslag inom programmeringsvärlden och en lång och stabil utvecklingshistorik.

5.3 Standarder för säker programmering

Secure Coding [14] är ett initiativ från Carnegie Mellon-universitetets CERT med målet att reducera antalet svagheter i programvara till en nivå som kan hanteras i operativa miljöer. Reduceringen ska uppnås genom att minska antalet programmeringsfel och genom att upptäcka och eliminera säkerhetsbrister under implementation och testning.

Inom Secure Coding-initiativet har det utvecklats en rad standarder för säker programmering och publicerats en rad böcker om hur säker programmering i olika programspråk går till. Exempel på standarder är *The CERT C Coding Standard* och *The CERT Oracle Secure Coding Standard for Java*. En lista med standarder och böcker finns tillgänglig på Carnegie Mellon CERT:ens webbsida[6].

MISRA C [64] är en standard för säker C-programmering av inbyggda system inom fordonsindustrin utvecklad av Motor Industry Software Reliability Association (MISRA). Användningen av standarden har spridit sig och den används numera även inom till exempel försvars-, rymd-, järnvägs- och medicinskteknisk industri. Första versionen av standarden utkom 1998 och uppdaterades 2004 (MISRA C:2004) i och med att en stor mängd förbättringsförslag inkommit sedan starten. Framför allt förbättrades överensstämmelsen med C90 och tvetydigheter eliminerades. Två korrigeringar av standarden publicerades 2007 innan version tre av standarden, MISRA C:2012, kom. De största förändringen är att även C99 nu följs. Dessutom ökade antalet kodexempel och de gjordes tydligare.

Open Web Application Security Project (OWASP) [78] ger ut en checklista med krav [79] med syftet att främja säker programmering inom främst webbtillämpningar. Kraven är programspråksoberoende och kräver programmeringskunskap vid användning. OWASP rekommenderar även tillgång till någon med IT-säkerhetskompetens i de projekt som använder kraven, även om dessa i sig inte kräver sådan kompetens vid användning. Personen med IT-säkerhetskompetens ska verka sammanhållande och se till att projektets IT-säkerhetsarbete hänger ihop.

Den version av OWASP:s krav som är aktuell 2016 är version 2 från 2010 och omfattar 17 sidor. Kraven är allmänt skrivna för att vara generellt tillämpliga och ger inte några detaljerade instruktioner om implementation. Följande exempel på krav är taget från delen om kryptografiska funktioner [79, sid. 9] och visar detaljnivån på kraven.

- *All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system (e.g., The server)*
- *Protect master secrets from unauthorized access*
- *Cryptographic modules should fail securely*
- *All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator when these random values are intended to be un-guessable* [79, sid. 9]

[6] `https://www.cert.org/secure-coding/standards/index.cfm`

Det finns även kommersiella företag som driver projekt kring säker programmering. Microsoft har ett projekt de kallar Security Development Lifecycle (SDL) [63] och Apple har sammanställt en checklista [42] för säkrare programmering.

5.4 Sårbarhetsförutsägelse i kod

För att undvika begränsningen det innebär att vara hänvisad till ett fåtal säkra programspråk görs försök med att tillåta programmering i valfritt programspråk och istället uppskatta antalet potentiella säkerhetsrisker i koden baserat på olika parametrar. Detta koncept kallas *sårbarhetsförutsägelse* och kan användas för att både minska utvecklingskostnaderna och öka pålitligheten hos det färdiga programmet. Konceptet kan liknas vid den mer etablerade tekniken att uppskatta antalet fel i koden genom att mäta olika egenskaper hos koden. Skillnaden är att inte alla fel innebär en sårbarhet. Genom att koncentrera sig på de sårbarhetsgivande felen kan koden göras säkrare med en mindre arbetsinsats än om alla fel skulle åtgärdas.

Inom den akademiska världen undersöks ett flertal olika metoder för förutsägelse av sårbarheter i kod. Metoderna baserar sig på mätning av allmänna kodegenskaper, till exempel olika komplexitetsmått, bland annat cyklomatisk komplexitet[7], nästlingsdjup[8], kodändringsfrekvens (*code churn*) och utvecklarens aktivitetsnivå. Forskning visar att metoderna ger tillfredsställande resultat. De är dock ännu inte tillräcklig mogna för att kunna användas kommersiellt.

Enligt en artikel från 2007 om sårbarhetsförutsägelser [3] utgjorde andelen sårbarheter ungefär 1–5 % av andelen fel i kod, något som också förutspåddes av experter inom området. Artikelförfattarna använde olika mätmetoder för att undersöka om antalet oupptäckta sårbarheter i kod kunde förutsägas. De observerade att sårbarhetsdensiteten hos programvara hamnade inom ett begränsat spann och att likartad programvara har likartad sårbarhetsdensitet. De noterade även att serveroperativsystem hade en högre andel sårbarheter i förhållande till fel än vanliga operativsystem.

I en samling empiriska studier [90] av mätmetoder för att bedöma andelen sårbarheter i programkod redan tidigt i utvecklingsprocessen konstaterades att de studerade metoderna visserligen fungerade, men att de fortfarande hade utvecklingspotential. Resultatet visade att cirka 70 % av de kända sårbarheter som fanns i koden kunde förutsägas genom mätning på mindre än 30 % av koden. I studierna användes öppen källkod (Mozilla Firefox och kärnan till Red Hat Enterprise Linux) tagen från internet och resultaten av mätningarna jämfördes med inrapporterade sårbarheter för källkoden. Artikelförfattarna testade även olika kom-

[7] Cyklomatisk komplexitet mäter antalet vägar genom koden som kan tas vid exekvering, det vill säga i grova drag antalet villkorssatser i koden. [57, 61]

[8] Nästlingsdjup mäter antalet nästlade loopar i koden. En loop inuti en annan loop ger högre värde än två loopar efter varandra. [57]

binationer av mätmetoder i ett försök att hitta den optimala kombinationen av algoritmer.

En svaghet hos sårbarhetsförutsägande mätmetoder är att de generellt har en hög andel falska positiv[9]. I en artikel [67] från 2015 undersöktes orsaken till varför modeller för sårbarhetsförutsägelse inte användes inom Microsoft i samma utsträckning som modeller för felförutsägelse gjorde. Undersökningen baserades på kodbasen till operativsystemet Windows. Artikelförfattarna konstaterade att mängden kod som behövde analyseras för hand och användas för träning av algoritmerna för att antalet falska positiv skulle hamna på praktiskt användbara nivåer blev för stor för att kunna hanteras inom rimlig tid. De drog även slutsatsen att förutsägelse på binärkodsnivå inte inte gav tillräckligt pålitliga resultat på källkodsnivå. De föreslog att fler säkerhetsspecifika mätmetoder och kombinationer av dem skulle undersökas i ett försök att förbättra resultaten, men förklarade även att det finns en gräns för vad som kan göras utan att lägga till domänspecifik kunskap i detektionsalgoritmerna.

En forskargrupp utvecklade 2015 utvecklingsverktyget VCCFinder [81] i ett försök att förbättra det existerande utvecklingsverktyget Flawfinder. VCCFinder visade sig i tester ha drygt 99 % färre[10] falska positiv än Flawfinder. Detta vid en detektionsgrad på drygt 24 % (53 av 219 kända fel) för båda utvecklingsverktygen. För att lyckas med det använde författarna meta-data insamlade från olika källkodslager på internet som en utökning till mätunderlaget. Utvecklingsverktyget tränades med hjälp av redan rapporterade sårbarheter för att lära sig identifiera nya.

[9] Falska positiv är resultat där algoritmen reagerar på någonting på felaktiga grunder, det vill säga falsklarm.

[10] Enligt artikeln gav VCCFinder 36 falska positiv på de data som användes, till skillnad från Flawfinders 5460

6 Säkra operativsystem

Ett säkert operativsystem (*Trusted Operating System*) innehåller i allmänhet fyra komponenter [49]:

Informationsuppdelning begränsar vilken information ett program får tillgång till.

Rolluppdelning begränsar vilken behörighet en användare har.

Minsta behörighet begränsar vad processer kan göra. Processer ska bara ges tillräckligt med rättigheter för att göra det de ska och inga rättigheter utöver det.

Säkerhetsbeslut på kernelnivå ser till att sådana beslut fattas på en låg nivå där program och användare inte kan påverka dem.

För att vara ett säkert operativsystem måste det också finnas bevis för riktighet. I militära system används även olika sekretessnivåer på informationen och därför ska också säkra operativsystem ge tillräckligt stöd för flernivåsäkerhet.

Det mest använda utvecklingsverktyget för att bevisa riktighet i programkod är CC med SFR:er för Labeled Security Protection Profile (LSPP) och Mandatory Access Control (MAC). I [83] listas bland annat de operativsystem som godkänts i kronologisk ordning, tillsammans med EAL-nivå och vilket land de godkänts i.

MAC definieras av USA:s DoD i The Orange Book som:

> *a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.* [103, sid. 109]

I FOI-rapporten FOI-R--4010--SE definieras MAC som:

> *MAC innebär att reglerna i modellen tillämpas på åtkomsträttigheter där användarna inte kan påverka sina egna rättigheter eller ändra rättigheter för andra användare. I ett MAC-system måste rättighetstilldelningen skötas av någon administratörsfunktion som är ”utanför systemet”.* [35, sid. 33]

I ett MAC-system fattas även besluten om åtkomstkontrolltilldelning av en central instans och hela åtkomstkontrollprocessen ligger således utanför användarnas kontroll. MAC-riktlinjer kan användas i till exempel flernivåsäkerhet, upprätthållande av typning och rollbaserad åtkomstkontroll.

En kort sammanfattning av LSPP ges i beskrivningen av PP-dokumentet:

> *LSPP conformant products support access controls that are capable of enforcing access limitations on individual users and data objects. Specifically, two classes of access control mechanisms are provided: those that allow individual users to specify how resources (e.g., files, directories) under their control are to be shared; and those that enforce limitations on sharing among users. The latter is implemented by the use of security markings (i.e., "labels"). LSPP-conformant products also provide an audit capability which records the security-relevant events which occur within the system.* [55]

Nedan ges en kort beskrivning av några säkra operativsystem.

6.1 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) [76] är en säkerhetsmodul till Linuxkärnan som ger en mekanism för säkerhetsriktlinjer för åtkomstkontroll som bland annat inkluderar MAC. Denna säkerhetsmodul kan adderas till olika Linuxdistributioner. Den utvecklades ursprungligen av USA:s National Security Agency (NSA).

SELinux i sig är inte ett säkert operativsystem, men tillför den viktiga säkerhetsfunktionen MAC som behövs för att bygga ett säkert operativsystem. Olika Linuxdistributioner har certifierats enligt LSPP i CC och de innehåller alla SELinux. Ett exempel är Red Hat Enterprise Linux 5, som är EAL 4+ certifierat mot LSPP.

6.2 Trusted Berkeley Software Distribution

Trusted Berkeley Software Distribution (BSD) [104] liknar SELinux, men är implementerad för operativsystemet BSD. Det innehåller bland annat stöd och funktionalitet för följande säkerhetsfunktioner:

- utökade attribut
- obligatoriska åtkomstkontrollistor
- filsystemet Unix File System 2 (UFS2)
- kontroll av säkerhetshändelser

- Basic Security Module (BSM)[1]-modulen OpenBSM [107]
- Pluggable Authentication Module (PAM)[2]-modulen OpenPAM [93]
- ett flexibelt ramverk för åtkomst till kärnan.

Säkerhetsfunktionerna i TrustedBSD har utvecklats sedan 2000 och återfinns numera i flera andra operativsystem, bland andra Linux samt Apples Mac OS X och iOS. Åtminstone fram till 2012 underhölls och vidareutvecklades funktionerna för operativsystemet FreeBSD.

6.3 STOP OS

STOP OS [5, 6] har certifierats för assuransnivå EAL 4+ enligt CC mot LSPP under villkor att det körs på fysisk eller virtualiserad x86-hårdvara. STOP OS är ett generellt flernivåsäkerhetsoperativsystem som tillåter flera användare samt multikörning och fungerar i nätverksmiljöer. Det är ungefär lika snabbt som Linux och har även ett Linux-API för att kunna köra Linux-programvara. STOP OS används i BAE System:s system XTS Guard 5 [115], vilken kan fungera som en datadiod.

Det som skiljer STOP OS från andra certifierade operativsystem är först och främst dess obligatoriska känslighets- (*mandatory sensitivity policy*) och integritetspolicy (*mandatory integrity policy*). Dessa två samverkar för att ge ett heltäckande skydd av data. Känslighetspolicyn är till för att förhindra obehörigt avslöjande av data. Integritetspolicyn ska förhindra obehörig radering eller ändring av data. Som grund för de två policyerna används de formella säkerhetsmodellerna Bell-LaPadula och Biba[3]. Den obligatoriska sekretesspolicyn verkställer amerikanska försvarsministeriets klassificeringsmodell för informationskänslighet (det vill säga Confidential, Secret och Top Secret).

Några svagheter jämfört med andra operativsystem som är certifierade på lägre EAL-nivåer är sämre prestanda, inget grafiskt användargränssnitt och begränsat urval av hårdvara.

6.4 INTEGRITY-178B

Realtidsoperativsystemet INTEGRITY-178B [44] är det enda operativsystemet som har utvärderats till EAL 6+ High Robustness mot U.S. Government Pro-

[1] BSM används i vissa operativsystem för att underlätta eller möjliggöra säkerhetsrevision. Det innehåller bland annat funktioner för att göra systemanrop och hantera revisionsposter. Det finns även ett speciellt filsystem som ger spårbarhet. BSM togs ursprungligen fram av Sun.

[2] PAM är en mekanism för att knyta samman olika autentiseringsmoduler på låg nivå i ett standardiserat format på hög nivå. PAM föreslogs av Sun 1995 och har sedan dess integrerats i ett flertal av de vanligare operativsystemen i Unix-familjen.

[3] En närmare presentation av säkerhetsmodellerna Bell-LaPadula och Biba, tillsammans med några andra liknande modeller, återfinns i FOI-rapporten FOI-R--4010--SE.

tection Profile for Separation Kernels in Environments Requiring High Robustness [106]. Det är egentligen bara INTEGRITY-178B Separation Kernel som har evaluerats. TOE:n exkluderar alltså komponenter som filsystemet och nätverkskomponenter som inkluderas i TOE av andra operativsystem (så som Windows eller Linux).

Till skillnad från LSPP som skyddar en ”antaget icke-fientlig och väladministrerad användargrupp som behöver skydd mot hot från oavsiktliga eller osofistikerade försök att kringgå systemets säkerhetsfunktioner”[4], så ger U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness skydd mot ”väl underbyggda sofistikerade attacker” [44].

[4]”assumed non-hostile and well-managed user community requiring protection against threats of inadvertent or casual attempts to breach the system security” [55]

7 Diskussion och slutsatser

Detta kapitel inleds med en diskussion av de tidigare delarna av rapporten. Diskussionen mynnar ut i att ett antal slutsatser dras, vilka leder vidare till förslag på framtida arbete för att ytterligare bredda och fördjupa kunskapen som ligger till grund för rapporten.

Under arbetet med att söka efter och sammanställa informationen som låg till grund för denna rapport framträdde ett antal trender vad gäller användning och utveckling av pålitlighetshöjande verktyg. I och med att en iterativ process användes vid informationssökningen har det inte gått att visa på statistiskt säkerställda resultat för dessa trender, underlaget var inte heltäckande och urvalet har påverkats av författarna. Det som redovisas är därför författarnas egna uppfattningar.

7.1 Diskussion

Inom de branscher där liv står på spel har användningen av pålitlighetshöjande verktyg kommit längre än inom IT-säkerhetsområdet. En tydlig indikator är det faktum att de exempel på praktisk användning av de olika pålitlighetshöjande verktyg som hittats alla handlar om transportindustrin. Det kan kännas naturligt att så är fallet, ett liv går egentligen inte att värdera rent ekonomiskt och måste det göras blir värdet mycket högt.

Traditionellt sett har inte IT-system haft någon direktverkande skadlig effekt på mänskligt liv. Detta är dock på väg att förändras i och med en ökande digitalisering och integrering av automatiska system i samhället. Haverier i IT-system kan i och med detta ge sekundäreffekter där liv mycket väl kan stå på spel. Således borde intresset för pålitlighetshöjande åtgärder vid systemutveckling öka även inom mer klassiska IT-branscher. Så verkar också vara fallet. Under arbetet med rapporten har författarna sett tendenser till att IT-branschen faktiskt uppfattat behovet. Ett tydligt tecken är de olika organisationer och initiativ till säker programmering som startats. De flesta stora IT-företag och organisationer driver sådana projekt, men det finns även ideella initiativ, till exempel OWASP och TSI. Behovet av pålitliga IT-system ökar, men frågan är om de ansträngningar som görs vad gäller forskning och utveckling kring pålitlighetshöjande verktyg är tillräckliga för att hålla jämn takt med samhällsutvecklingen. Författarna uppfattar det som att mycket görs, men att samordning saknas och att branschens beteende än så länge är ganska yrvaket. Likaså står den för närvarande populära agila utvecklingsmetodiken i bjärt kontrast till den mer styrda metodik som krävs för att ge pålitlighet. Sökandet efter den optimala utvecklingsmetodiken verkar ha missat parametern pålitlighet, något som förmodligen skulle kunna lösas med mer samverkan mellan de ingående forskningsområdena.

Användning av formella metoder har potential att ge stora bidrag till arbetet med pålitlighet i IT-system, men lider av omognad, ad hocmentalitet och annorlunda kompetenskrav relativt traditionell utvecklingsmetodik. Formella metoder ger dock möjlighet till en heltäckande och anpassad pålitlighetslösning, men har betraktats som svåranvända och med krav på matematisk kompetens hos utvecklarna. De åsikterna verkar dock ha börjat luckras upp, förmodligen beroende på flera orsaker. För det första har utvecklingen av ramverk för att underlätta användning av olika formella metoder kommit så långt inom forskarvärlden att det har börjat dyka upp kommersiella utvecklingsverktyg baserade på formella metoder. För det andra drivs utvecklingen på av att flera standarder innehåller krav på användning av formella metoder. Även köparnas ökande krav på certifiering av IT-produkter gynnar utvecklingen mot ökad pålitlighet.

Ingen kedja är dock starkare än dess svagaste länk. Detta gäller även vid utveckling av pålitliga system. Pålitlighetshöjande verktyg måste användas genom hela utvecklingsprocessen för att maximal pålitlighet ska kunna uppnås. Tyvärr är täckningsgraden för de verktyg som finns inte fullständig och i många fall krävs ad hoclösningar och specialanpassningar i vissa steg av utvecklingsprocessen. Likaså innebär användningen av pålitlighetshöjande verktyg krav på delvis nya kompetenser jämfört med traditionell utvecklingsmetodik. Det går dock att använda utvecklingsverktygen under enbart delar av utvecklingsprocessen. Varje användning av pålitlighetshöjande verktyg, hur liten den än är, förbättrar resultatet. Tyvärr tenderar användningen att bli precis så stor som krävs för att att uppfylla kundens krav. Om kunden inte har kompetens och förmåga att ställa nödvändiga och tillräckliga krav blir resultatet att pålitligheten hos slutprodukten brister. Det krävs således en generell kompetenshöjning vad gäller pålitlighetsfrämjande verktyg både på kund- och leverantörssidan.

I ett försök att sänka trösklarna för användning av pålitlighetshöjande verktyg har det utvecklats säkra programspråk och säkra operativsystem som är tänkta att (till viss del) avlasta systemutvecklarna. Dessa programspråk och operativsystem är dock bara delar i en pålitlighetslösning och bör enbart ses som nödvändiga men inte tillräckliga delar. Detta på grund av att det fortfarande går att skriva kod som de facto gör fel även i ett säkert programspråk. Den fördel ett sådant språk ger är skydd mot vanliga buggar och informationsläckage. Detsamma gäller för säkra operativsystem, vars främsta egenskap är att de har pålitliga säkerhets- och separationsegenskaper. I och med att dessa egenskaper är inbyggda i programspråket respektive operativsystemet underlättar de för programmeraren att skriva säker programkod, men begränsar möjligheterna till fria val av implementationslösningar, vilket kan vara som ett hinder. Även om det påtvingade arbetssättet kan upplevas vara begränsande och mer arbetskrävande är vinsten för kvaliteten på slutprodukten större. Dock är användning av sådana verktyg inte någon mirakelkur, utan bör kombineras med andra tekniker, till exempel formella metoder, i hela utvecklingsprocessen.

För närvarande är administrativa verktyg såsom standarder, certifiering med mera det som främst används för att uppnå pålitlighet. Det är dock en enkel och inte alltid så verkningsfull lösning. Kritik riktas ofta mot att standarderna är fokuserade på dokumentation och inte det praktiska programmeringsarbetet. Å andra sidan ökar den upplevda pålitligheten hos produkterna och till viss del även den faktiska. För att så inte ska vara fallet måste utvecklarna medvetet fuska, säga en sak och göra en annan. Faktum kvarstår dock, det finns inte någon garanti för att det som skrivs i dokumenten görs i verkligheten. I till exempel CC krävs dock användning av formella metoder i utvecklingsprocessen vid klassning i högre säkerhetsnivåer.

Förmågan att enkelt mäta antalet säkerhetskritiska fel, eller i alla fall uppskatta risken för dem, är något som förenklar utveckling av pålitlig programvara. Valfritt och för tillämpningen lämpligt programspråk kan användas och mätningen ger ett faktiskt värde för kvaliteten på koden, till skillnad från andra pålitlighetshöjande metoder där kvalitetsmåttet tas fram genom en bedömning av utvecklingsmiljön och -processen. Tyvärr är forskningen inte där ännu. De resultat för sårbarhetsförutsägelse i kod vi sett är knappast acceptabla och än mindre användbara i praktiken. De måste förbättras många magnituder innan de kan användas praktiskt. Området som sådant är dock intressant och vissa metoder kan fortfarande användas som grova indikatorer på kodkvaliteten.

Forskningsområdet kring pålitlighet i programvara är stort och omfattande, med en brokig skara av specialområden. Forskningen kring formella metoder får ses som kärnan i huvudområdet. Vad vi kunnat se har övriga forskningsområden kring pålitlighet i programvara någon slags koppling till formella metoder. Områdena är dock mer eller mindre isolerade från varandra. Vi har inte sett några gränsöverskridande artiklar där flera forskargrupper samarbetat, utan standard är att bygga vidare på det tidigare arbete som gjorts i den egna forskargruppen. Mycket arbete bygger till exempel på Z- och B-språken, men knappt på någonting utanför forskningsområdet kring formella metoder.

Försvarsmakten visar ett stort intresse för att använda pålitlighetshöjande verktyg och ursprunget till den här studien är just det intresset. Pålitlighetshöjande verktyg har ett självklart användningsområde inom Försvarsmaktens verksamhet. Dels höjer det värdet vid användning av befintliga system, dels kan det i förlängningen radikalt minska antalet säkerhetskritiska fel ur Försvarsmaktens IT-system. Det i sin tur innebär att även civila system i framtiden kan dra nytta av metodiken och på så sätt höja den samhällsövergripande IT-säkerheten, vilket är av vikt i vårt allt mer digitaliserade samhälle.

7.2 Slutsatser

Trots att pålitlighetshöjande verktyg av olika slag funnits i närmare 50 år har de inte fått något egentlig praktiskt genomslag inom programutvecklingsbranschen. På senare tid har dock detta börjat förändras och utvecklingen verkar acce-

lerera, vilket är positivt. Det är dock långt kvar innan ett helt pålitligt IT-system ser dagens ljus. Än så länge handlar det om att minska sannolikheten för fel.

Pålitlighet i programvara är ett viktigt och naturligt steg mot nästa nivå av säker och pålitlig programvara. Det är därför ett forskningsområde som mycket väl behövs och som definitivt är värt att studera vidare, speciellt med tanke på nyttan för Försvarsmakten.

7.3 Framtida arbete

Det finns en mängd forskningsfrågor kring pålitlighet i programvara att besvara i kommande projekt. Ett av dem är en fördjupning i metodik för att kunna mäta och bedöma sannolikheten för att programkod innehåller säkerhetskritiska fel och brister. En möjlighet till automatisering och snabb objektiv bedömning av sådana parametrar skulle kunna öka pålitligheten hos programvaran, korta ledtiden och sänka kostnaderna vid programutveckling.

Ytterligare ett område som behöver studeras vidare är praktiska tillämpningar av formella metoder för programutveckling. Det är speciellt studier av svårigheter och utmaningar vid användning av pålitlighetshöjande verktyg som behöver göras. Försvarsmakten har behov av praktisk erfarenhet av användning av formella metoder vid programutveckling för att kunna bedöma om de kan användas inom verksamheten. Många av deras verksamheter skulle vara betjänta av att bättre känna till metodernas styrkor och svagheter ur en praktisk synvinkel.

Även forskningen kring sårbarhetsförutsägelse är intressant att studera mer. Utvecklingen verkar ha accelererat på senare tid och konceptet är användbart för till exempel grovsortering av programkod. Kod som inte klarar satta gränser för antal sårbarheter vid sårbarhetsförutsägelse kan direkt avfärdas, medan de som klarar gränsen får studeras närmare med andra metoder. Tekniken bakom mätmetoderna som tagits fram är också användbara i andra sammanhang och fortsatta studier skulle därför kunna ge synergieffekter.

Litteratur

[1] *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. Läst 2016-03-01. URL: http://cwe.mitre.org/top25/.

[2] A. Adams. *Meta-theory in the Higher-Order Logic Framework Isabelle*. Läst 2016-05-25. Jan. 1996. URL: https://rd.host.cs.st-andrews.ac.uk/publications/Isabelle.ps.

[3] O. H. Alhazmi, Y. K. Malaiya och I. Ray. "Measuring, Analyzing and Predicting Security Vulnerabilities in Software Systems". I: *Comput. Secur.* 26.3 (maj 2007), s. 219–228. DOI: 10.1016/j.cose.2006.10.002.

[4] J. B. Almeida, M. J. Frade, J. S. Pinto och S. M. de Sousa. *Rigorous Software Development – An Introduction to Program Verification*. Springer-Verlag London, 2011. DOI: 10.1007/978-0-85729-018-2.

[5] BAE Systems. *STOP OS^{TM}*. Läst 2015-11-24. 2015. URL: http://www.baesystems.com/en/product/stop-ostrade.

[6] BAE Systems. *STOP OS^{TM} Version 7.3.1 Security Target*. Läst 2015-11-24. Dec. 2011. URL: https://www.commoncriteriaportal.org/files/epfiles/BAE%20STOP%207%20Security%20Target%201.08.pdf.

[7] G. Barthe, P. Courtieu, G. Dufay och S. Melo de Sousa. "Tool-Assisted Specification and Verification of Typed Low-Level Languages". I: *Journal of Automated Reasoning* 35.4 (2006), s. 295–354. DOI: 10.1007/s10817-005-0084-6.

[8] P. Baudin, F. Bobot, R. Bonichon, L. Correnson, P. Cuoq, Z. Dargaye, J.-C. Filliâtre, P. Herrmann, F. Kirchner, M. Lemerre, C. Marché, B. Monate, Y. Moy, A. Pacalet, V. Prevosto, J. Signoles och B. Yakobowski. *Frama-C*. Läst 2016-03-24. Febr. 2016. URL: http://frama-c.com/index.html.

[9] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy och V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. Tekn. rapport Version 1.10. Läst 2016-03-24. CEA LIST och INRIA, 2013.

[10] J. Berg och B. Jacobs. "The loop Compiler for Java and JML". I: *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings*. Utg. av T. Margaria och W. Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, s. 299–312. DOI: 10.1007/3-540-45319-9_21.

[11] P. Blackburn, J. Bos och K. Striegnitz. *Learn Prolog Now!* Läst 2016-03-16. 2012. URL: http://www.learnprolognow.org/.

[12] *Build Security In project.* Läst 2016-03-01. URL: https://buildsecurityin.us-cert.gov/.

[13] R. M. Burstall och J. A. Goguen. "The Semantics of CLEAR, A Specification Language". I: *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School.* Läst 2016-03-02. London, UK, UK: Springer-Verlag, 1980, s. 292–332. URL: http://dl.acm.org/citation.cfm?id=647448.727240.

[14] Carnegie Mellon University. *Secure Coding Initiative.* Läst 2016-04-15. URL: https://www.cert.org/secure-coding/.

[15] *CESG Commercial Product Assurance (CPA).* Läst 2015-11-23. URL: https://www.cesg.gov.uk/servicecatalogue/Product-Assurance/CPA/Pages/CPA.aspx.

[16] *CESG Tailored Assurance Service.* Läst 2016-03-01. URL: https://www.cesg.gov.uk/scheme/tailored-evaluation.

[17] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe och G. Smeding. *Reference Manual of the LNT to LOTOS Translator.* Tekn. rapport. Läst 2016-03-03. Inria Grenoble - Rhône-Alpes/CONVECS, 2010. URL: ftp://ftp.inrialpes.fr/pub/vasy/publications/cadp/Champelovier-Clerc-Garavel-et-al-10.pdf.

[18] L. Chen. *Direct Anonymous Attestation (DAA).* Läst 2015-12-18. Okt. 2005. URL: https://www.trustedcomputinggroup.org/files/resource_files/AC0E3E05-1D09-3519-ADCD93471A61681A/051012_DAA-slides.pdf.

[19] I. Claßen. "Revised ACT ONE: Categorical constructions for an algebraic specification language". I: *Categorical Methods in Computer Science With Aspects from Topology.* Utg. av H. Ehrig, H. Herrlich, H. Kreowski och G. Preuß. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, s. 124–141. DOI: 10.1007/3-540-51722-7_8.

[20] I. Claßen, H. Ehrig och D. Wolz. *Algebraic Specification Techniques and Tools for Software Development: The ACT Approach.* River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1993.

[21] ClearSy System Engineering. *Presentation of the B Method.* Läst 2016-03-24. 2015. URL: http://www.methode-b.com/en/b-method/.

[22] CoFI. *CASL – From CoFI.* Läst 2016-03-02. Febr. 2008. URL: http://www.informatik.uni-bremen.de/cofi/index.php/CASL.

[23] E. Cohen. *VCC.* Läst 2016-03-24. Aug. 2012. URL: https://vcc.codeplex.com/.

[24] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte och S. Tobies. "VCC: A Practical System for Verifying Concurrent C". I: *Theorem Proving in Higher Order Logics: Proceedings of the 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009.* Utg. av S. Berghofer, T. Nipkow, C. Urban och M. Wenzel. Lecture Notes in Computer Science (LNCS) 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 23–42. DOI: 10.1007/978-3-642-03359-9_2. URL: http://research.microsoft.com/en-us/um/people/moskal/pdf/tphol2009.pdf.

[25] E. Cohen, M. Hillebrand, M. Moskal, W. Schulte och S. Tobies. *Verifying Concurrent C Programs with VCC.* Tekn. rapport Working draft, ver. 0.2. Microsoft, maj 2012. URL: http://research.microsoft.com/en-us/um/people/moskal/pdf/vcc-tutorial-col2.pdf.

[26] CVC3. *Home.* Läst 2016-03-24. URL: http://www.cs.nyu.edu/acsys/cvc3/.

[27] CVC4. *the smt solver.* Läst 2016-03-24. 2016. URL: http://cvc4.cs.nyu.edu/web/.

[28] M. Davis, G. Logemann och D. Loveland. "A Machine Program for Theorem-proving". I: *Commun. ACM* 5.7 (juli 1962), s. 394–397. DOI: 10.1145/368273.368557.

[29] M. Davis och H. Putnam. "A Computing Procedure for Quantification Theory". I: *J. ACM* 7.3 (juli 1960), s. 201–215. DOI: 10.1145/321033.321034.

[30] D. E. Denning och P. J. Denning. "Certification of Programs for Secure Information Flow". I: *Communications of the ACM* 20.7 (juli 1977), s. 504–513. DOI: 10.1145/359636.359712.

[31] D. Detlefs, G. Nelson och J. Saxe. *Simplify: A Theorem Prover for Program Checking.* Läst 2016-03-24. Juli 2003. URL: http://www.hpl.hp.com/techreports/2003/HPL-2003-148.pdf.

[32] Department of Homeland Security (DHS). *DHS Software Assurance Program.* Läst 2016-03-01. URL: https://buildsecurityin.us-cert.gov/software-assurance.

[33] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasreanu, R. H. Zheng och W. Visser. "Tool-supported program abstraction for finite-state verification". I: *Proceedings of the 23rd International Conference on Software Engineering, 2001. ICSE 2001.* Maj 2001, s. 177–187. DOI: 10.1109/ICSE.2001.919092.

[34] ECMA International. *Eiffel:Analysis, Design and Programming Language.* Läst 2016-03-24. Juni 2006. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf.

[35] D. Eidenskog och L. Westerdahl. *Metoder för informations- och åtkomstkontroll*. Tekn. rapport FOI-R--4010--SE. Informationssäkerhet och IT-arkitektur, 2014.

[36] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe och R. Stata. "Extended Static Checking for Java". I: *SIGPLAN Not.* 37.5 (maj 2002), s. 234–245. DOI: 10.1145/543552.512558.

[37] *Flow Caml*. Läst 2016-03-01. URL: http://www.normalesup.org/~simonet/soft/flowcaml/.

[38] *Free Software and Open Source Development with Ada*. Läst 2015-11-23. URL: http://libre.adacore.com/.

[39] J. Goguen. *The OBJ Family*. Läst 2016-03-02. Dec. 2005. URL: http://cseweb.ucsd.edu/~goguen/sys/obj.html.

[40] J. Guttag, J. Horning och A. Modet. *Report on the Larch Shared Language – Version 2.3*. Tekn. rapport SRC Research Report 58. Läst 2016-03-02. Digital Equipment Corporation, 1990. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-58.pdf.

[41] D. Hedin. "A Perspective on Information-Flow Control". I: *The 2011 Marktoberdorf Summer School*. 2011.

[42] A. Inc. *Secure Coding Guide*. Läst 2016-04-14. Febr. 2014. URL: https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/SecurityDevelopmentChecklists/SecurityDevelopmentChecklists.html.

[43] INRIA. *The Coq Proof Assistant*. Läst 2016-03-24. Jan. 2016. URL: https://coq.inria.fr/.

[44] *INTEGRITY-178B Real Time Operating System (RTOS)*. Läst 2016-03-01. URL: http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[45] *Isabelle*. Läst 2016-05-25. Jan. 2016. URL: https://isabelle.in.tum.de/.

[46] ISO. *ISO 8807:1989 – Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. Tekn. rapport. International Organization for Standardization (ISO), 1989.

[47] ISO. *ISO/IEC 15437:2001 – Information technology – Enhancements to LOTOS (E-LOTOS)*. Tekn. rapport. International Organization for Standardization (ISO), 2001.

[48] W. Jackson. *Under attack: common criteria has loads of critics, but is it getting a bum rap?* Läst 2016-03-01. URL: http://gcn.com/articles/2007/08/10/under-attack.aspx.

[49] C. Jacobs. *Trusted Operating Systems.* Tekn. rapport. Läst 2016-03-01. 2001. URL: http://www.giac.org/paper/gsec/842/trusted-operating-systems/101762.

[50] *Jif.* Läst 2016-03-01. URL: http://www.cs.cornell.edu/jif/.

[51] M. Kaufmann och J. Moore. *ACL2 Version 7.2.* Läst 2016-03-16. Jan. 2016. URL: https://www.cs.utexas.edu/users/moore/acl2/.

[52] *The KeY Project.* Läst 2016-03-24. URL: http://www.key-project.org/.

[53] C. Koster. *ELAN educational language.* Läst 2016-03-16. URL: http://www.cs.ru.nl/elan/.

[54] Krisberedskapsmyndigheten. *Basnivå för informationssäkerhet (BITS).* Läst 2015-12-18. 2006. URL: http://rib.msb.se/Filer/pdf/24855.pdf.

[55] *Labeled Security Protection Profile.* Läst 2016-03-01. Okt. 1999. URL: http://www.commoncriteriaportal.org/files/ppfiles/lspp.pdf.

[56] M. Li och S. Liu. "SOFL Specification Animation with Tool Support". I: *Structured Object-Oriented Formal Language and Method: Third International Workshop, SOFL+MSVL 2013, Queenstown, New Zealand, October 29, 2013, Revised Selected Papers.* Utg. av S. Liu och Z. Duan. Vol. LNCS 8332. Springer International Publishing, 2014, s. 118–131. DOI: 10.1007/978-3-319-04915-1_9.

[57] A. Lindell. *En sammanställning av komplexitetsmått.* Läst 2016-05-11. Febr. 2012. URL: http://fileadmin.cs.lth.se/cs/education/EDA270/Reports/2012/Lindell.pdf.

[58] J. Löfvenberg och I. Rodhe. *Litteraturstudie av tekniker för pålitliga IT-plattformar.* Tekn. rapport FOI-R--3724--SE. Totalförsvarets forskningsinstitut (FOI), 2013.

[59] C. Marché. *The Krakatoa Verification Tool for JAVA programs.* Tekn. rapport Version 2.35. Läst 2016-03-24. INRIA Saclay - Île-de-France, mars 2015. URL: http://krakatoa.lri.fr/krakatoa.pdf.

[60] G. McGraw, S. Migues, J. West och B. Chess. *BSIMM.* Läst 2015-12-18. URL: https://www.bsimm.com/about/.

[61] Microsoft. *Code Metrics Values.* Läst 2016-05-11. 2016. URL: https://msdn.microsoft.com/en-us/library/bb385914.aspx.

[62] Microsoft. *Spec#.* Läst 2016-03-24. 2016. URL: http://research.microsoft.com/en-us/projects/specsharp/.

[63] Microsoft. *What is the Security Development Lifecycle?* Läst 2016-04-14. 2016. URL: https://www.microsoft.com/en-us/sdl/default.aspx.

[64] MISRA. *A brief history of MISRA C*. Läst 2016-04-14. URL: http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx.

[65] *Mission Statement*. Läst 2015-11-19. URL: http://www.uk-tsi.org/mission-statement.

[66] C. Mitchell. *Trusted Computing*. Computing and Networks Series. Institution of Engineering and Technology, 2005.

[67] P. Morrison, K. Herzig, B. Murphy och L. Williams. "Challenges with Applying Vulnerability Prediction Models". I: *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. HotSoS '15. Urbana, Illinois: ACM, 2015, 4:1–4:9. DOI: 10.1145/2746194.2746198.

[68] S. J. Murdoch, M. Bond och R. Anderson. "How Certification Systems Fail: Lessons from the Ware Report". I: *IEEE Security Privacy* 10.6 (nov. 2012), s. 40–44. DOI: 10.1109/MSP.2012.89.

[69] A. C. Myers och B. Liskov. "A decentralized model for information flow control". I: *The 16th ACM Symposium on Operating Systems Principles (SOSP)*. 1997, s. 129–142.

[70] *NASA-STD 8739.8 Standard for Software Assurance*. Läst 2016-03-01. URL: https://www.hq.nasa.gov/office/codeq/doctree/87398.pdf.

[71] *National Information Assurance (IA) Glossary*. Läst 2016-03-01. URL: http://www.ncsc.gov/nittf/docs/CNSSI-4009_National_Information_Assurance.pdf.

[72] Nationalencyklopedin (NE). *attestera*. Läst 2016-08-22. URL: http://www.ne.se/uppslagsverk/ordbok/svensk/attestera.

[73] Nationalencyklopedin (NE). *C A R Tony Hoare*. Läst 2015-11-02. URL: http://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/c-a-r-tony-hoare.

[74] Nationalencyklopedin (NE). *funktionella programspråk*. Läst 2015-09-29. URL: http://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/funktionella-programspr%C3%A5k.

[75] Nationalencyklopedin (NE). *lambdakalkyl*. Läst 2015-11-17. URL: http://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/lambdakalkyl.

[76] National Security Agency (NSA). *SELinux*. Läst 2016-03-01. URL: http://www.nsa.gov/research/selinux/.

[77] OCamlPro. *Alt-Ergo – An SMT Solver for Software Verification*. Läst 2016-03-24. 2016. URL: https://alt-ergo.ocamlpro.com/.

[78] OWASP. *About The Open Web Application Security Project*. Läst 2016-04-14. Febr. 2016. URL: https://www.owasp.org/index.php/About_OWASP.

[79] OWASP. *OWASP Secure Coding Practices Quick Reference Guide*. 2. utg. Läst 2016-04-14. Open Web Application Security Project (OWASP). Nov. 2010. URL: https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf.

[80] L. C. Paulson. "Natural deduction as higher-order resolution". I: *The Journal of Logic Programming* 3.3 (1986), s. 237–258. DOI: http://dx.doi.org/10.1016/0743-1066(86)90015-4.

[81] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl och Y. Acar. "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits". I: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA: ACM, 2015, s. 426–437. DOI: 10.1145/2810103.2813604.

[82] B. C. Pierce. *Types and Programming Languages*. MIT Press, febr. 2002.

[83] C. C. Portal. *Certified Products*. Läst 2015-11-24. URL: https://www.commoncriteriaportal.org/products.

[84] F. Pothon. *DO-178C/ED-12C versus DO-178B/ED-12B – Changes and Improvements*. Tekn. rapport. Läst 2016-03-24. ACG Solutions, sept. 2012. URL: http://www.adacore.com/uploads/technical-papers/DO178C-ED12C-Changes_and_Improvements-Sep2012.pdf.

[85] A. Price. *TRUSTED COMPUTING GROUP (TCG) TIMELINE*. Läst 2015-09-29. Febr. 2011. URL: https://www.trustedcomputinggroup.org/files/resource_files/B8FF1287-1A4B-B294-D0423684DEB619FD/TCG%20Timeline_rev%20Feb%202011.pdf.

[86] A. Rasoolzadegan och A. A. Barfourosh. "A New Approach to Software Development Process with Formal Modeling of Behavior Based on Visualization". I: *The Sixth International Conference on Software Engineering Advances (ICSEA 2011)*. Utg. av L. Lavazza, L. Fernandez-Sanz, O. Panchenko och T. Kanstrén. okt 2011, s. 104–111.

[87] J. Rushby. *Formal Methods and their Role in the Certification of Critical Systems*. Tekn. rapport SRI-CSL-95-1. Menlo Park, CA: Computer Science Laboratory, SRI International, mars 1995. URL: http://www.csl.sri.com/papers/csl-95-1/.

[88] A. Sabelfeld och A. C. Myers. "Language-based Information-flow Security". I: *IEEE J.Sel. A. Commun.* 21.1 (sept. 2006), s. 5–19. DOI: 10.1109/JSAC.2002.806121.

[89] A. Sabelfeld och D. Sands. "Declassification: Dimensions and principles". I: *Journal of Computer Security* 17.5 (2009), s. 517–548. URL: https://www.semanticscholar.org/paper/Declassification-Dimensions-and-principles-Sabelfeld-Sands/349842108aa31fdebcd01b58924ade3f125d3c6f/pdf.

[90] Y. Shin, A. Meneely, L. Williams och J. A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". I: *IEEE Transactions on Software Engineering* 37.6 (nov. 2011), s. 772–787. DOI: 10.1109/TSE.2010.81.

[91] M. Sighireanu. *LOTOS NT User's Manual (Version 2.8)*. Tekn. rapport. Läst 2016-03-03. Uppdateringar gjorda av Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe och Jan Stöcker. INRIA Rhône-Alpes/VASY, febr. 2016. URL: ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.pdf.

[92] V. Simonet. "Flow Caml in a Nutshell". I: *Proceedings of the first APPSEM-II workshop*. 2003, s. 152–165.

[93] D.-E. Smørgrav. *OpenPAM*. Läst 2016-04-15. Juni 2014. URL: http://www.openpam.org.

[94] *Software Assurance Metrics And Tool Evaluation project*. Läst 2016-03-01. URL: http://samate.nist.gov.

[95] *SPARK 2014*. Läst 2015-11-23. URL: http://www.spark-2014.org/.

[96] *SPARK Ada*. Läst 2015-11-23. URL: http://www.adaic.org/advantages/spark-ada/.

[97] *SPARK Pro*. Läst 2015-11-23. URL: http://www.adacore.com/sparkpro/.

[98] SRI International. *The Yices SMT Solver*. Läst 2016-03-24. Dec. 2015. URL: http://yices.csl.sri.com/.

[99] *The Caml Language*. Läst 2016-03-01. URL: http://caml.inria.fr.

[100] *The Fabric Language*. Läst 2016-03-01. URL: http://www.cs.cornell.edu/Projects/fabric.

[101] *The Swift Project*. Läst 2016-03-01. URL: http://www.cs.cornell.edu/jif/swift.

[102] *The Trustworthy Software Initiative*. Läst 2015-11-19. URL: http://www.uk-tsi.org.

[103] *Trusted Computer System Evaluation Criteria*. Läst 2015-11-19. URL: http://csrc.nist.gov/publications/history/dod85.pdf.

[104] *TrustedBSD Project*. Läst 2016-03-01. URL: http://www.trustedbsd.org.

[105] Trustworthy Software Initiative och British Standards Institution. *Software Trustworthiness – Governance and management – Specification*. Läst 2015-11-19. Juni 2014. URL: http://www.uk-tsi.org/pas754.

[106] *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. Läst 2016-03-01. URL: https://www.niap-ccevs.org/pp/pp_skpp_hr_v1.03.pdf.

[107] R. N. M. Watson. *OpenBSM: Open Source Basic Security Module (BSM) Audit Implementation*. Läst 2015-12-18. 2012. URL: http://www.trustedbsd.org/openbsm.html.

[108] *Welcome to* ERights.org*, home of E, the secure distributed persistent language for capability-based smart contracting*. Läst 2015-11-24. URL: http://erights.org/.

[109] *Welcome to the ERights.org wiki*. Läst 2015-11-24. URL: http://wiki.erights.org/wiki/Main_Page.

[110] R. Wieringa och E. Dubois. "Integrating Semi-Formal and Formal Software Specification Techniques". I: *Information Systems* 19.4 (1994). Läst 2015-10-06, s. 33–54. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=FD04653E9EB95EB68E1027622D781184?doi=10.1.1.53.9912&rep=rep1&type=pdf.

[111] R. Wieringa och E. Dubois. "Integrating semi-formal and formal software specification techniques". I: *Information Systems* 23.3–4 (1998). Återpublicering av en artikel från 1994 i Information Systems i ett specialnummer med titeln: Advance information systems engineering, s. 159–178. DOI: http://dx.doi.org/10.1016/S0306-4379(98)00007-6. URL: http://www.sciencedirect.com/science/article/pii/S0306437998000076.

[112] C. Wintersteiger. *Z3Prover/z3*. Läst 2016-03-24. Okt. 2015. URL: https://github.com/Z3Prover/z3/wiki#background.

[113] M. H. L. Wong Cheng In. "Informal, semi-formal, and formal approaches to the specification of software requirements". Läst 2015-10-06. magisteruppsats. University of British Columbia, Canada, nov. 1994. URL: https://circle.ubc.ca/bitstream/handle/2429/5607/ubc_1994-0660.pdf?sequence=1.

[114] J. Woodcock, P. G. Larsen, J. Bicarregui och J. Fitzgerald. "Formal Methods: Practice and Experience". I: *ACM Comput. Surv.* 41.4 (okt. 2009), 19:1–19:36. DOI: 10.1145/1592434.1592436.

[115] *XTS Guard 5*. Läst 2015-11-24. URL: http://www.baesystems.com/en/product/xts-guard-5.

[116] S. A. Zdancewic. "Programming Languages for Information Security". Diss. Cornell University, 2002. URL: https://www.cis.upenn.edu/~stevez/papers/Zda02.pdf.

A Förkortningar

Detta är en lista med förkortningar som används i rapporten. Efter varje förkortning i listan visas sidnumret för den första förekomsten av förkortningen i texten.

ACL2 A Computational Logic for Applicative Common Lisp . . . 18

ASM Abstract State Machine . . . 14

BIOS Basic Input/Output System . . . 30

BITS Basnivå för informationssäkerhet . . . 27

BSD Berkeley Software Distribution . . . 42

BSIMM Building Security In Maturity Model . . . 31

BSM Basic Security Module . . . 43

CASL Common Algebraic Specification Language . . . 15

CC Common Criteria for Information Technology Security Evaluation . . . 7

CCPA CESG Commercial Product Assurance . . . 25

CEM Common Evaluation Methodology . . . 25

CESG Communications-Electronics Security Group . . . 25

CIC Calculus of Inductive Construction . . . 20

CMVP Cryptographic Module Validation Program . . . 26

CTAS CESG Tailored Assurance Service . . . 25

CVC Cooperating Validity Checker . . . 18

DAA Direct Anonymous Attestation . . . 30

DHS Department of Homeland Security . . . 30

DoD Department of Defense . . . 23

EAL Evaluation Assurance Level . . . 11

E-LOTOS Enhanced LOTOS . . . 16

FIPS Federal Information Processing Standards . . . 25

IEEE Institute of Electrical and Electronics Engineers . . . 23

INRIA Institut national de recherche en informatique et en automatique . 16

LIS Ledningssystem för informationssäkerhet . . . 27

LOTOS Language Of Temporal Ordering Specification . . . 16

LSL Larch Shared Language . . . 15

LSPP Labeled Security Protection Profile 41
MAC Mandatory Access Control 41
MISRA Motor Industry Software Reliability Association 37
MSB Myndigheten för samhällsskydd och beredskap 27
NE Nationalencyklopedin 54
NIST National Institute of Standards and Technology 31
NSA National Security Agency 42
OWASP Open Web Application Security Project 37
PAM Pluggable Authentication Module 43
PLIT Pålitliga IT-plattformar 8
PP Protection Profile 24
PRIDE Profitable Information by Design 21
SAFECode Software Assurance Forum for Excellence in Code 30
SAMATE Software Assurance Metrics And Tool Evaluation 31
SAP Software Assurance Program 30
SAR Security Assurance Requirement 23
SCADE Safety-Critical Application Development Environment 14
SDM System Development Methodology 21
SDL Security Development Lifecycle 38
SELinux Security-Enhanced Linux 42
SFR Security Functional Requirement 23
SML Standard MetaLanguage 15
SMT Satisfiability Modulo Theories 18
SOFL Structure Object-oriented Formal Language 12
SSA Software Security Assessment 31
SSADM Structured Systems Analysis and Design Method 21
ST Security Target 24
TCG Trusted Computing Group 29
TCPA Trusted Computing Platform Alliance 29
TCSEC Trusted Computer System Evaluation Criteria 23
TIST Teknik för IT-säkerhet 8
TOE Target of Evaluation 23
TPM Trusted Platform Module 29

TSI Trustworthy Software Initiative 30
UFS2 Unix File System 2 42
UML Unified Modeling Language 11
VCC Verifying Concurrent C 19
VDM Vienna Development Method 14

FOI är en huvudsakligen uppdragsfinansierad myndighet under Försvarsdepartementet. Kärnverksamheten är forskning, metod- och teknikutveckling till nytta för försvar och säkerhet. Organisationen har cirka 1000 anställda varav ungefär 800 är forskare. Detta gör organisationen till Sveriges största forskningsinstitut. FOI ger kunderna tillgång till ledande expertis inom ett stort antal tillämpningsområden såsom säkerhetspolitiska studier och analyser inom försvar och säkerhet, bedömning av olika typer av hot, system för ledning och hantering av kriser, skydd mot och hantering av farliga ämnen, IT-säkerhet och nya sensorers möjligheter.

FOI
Totalförsvarets forskningsinstitut
164 90 Stockholm

Tel: 08-55 50 30 00
Fax: 08-55 50 31 00

www.foi.se