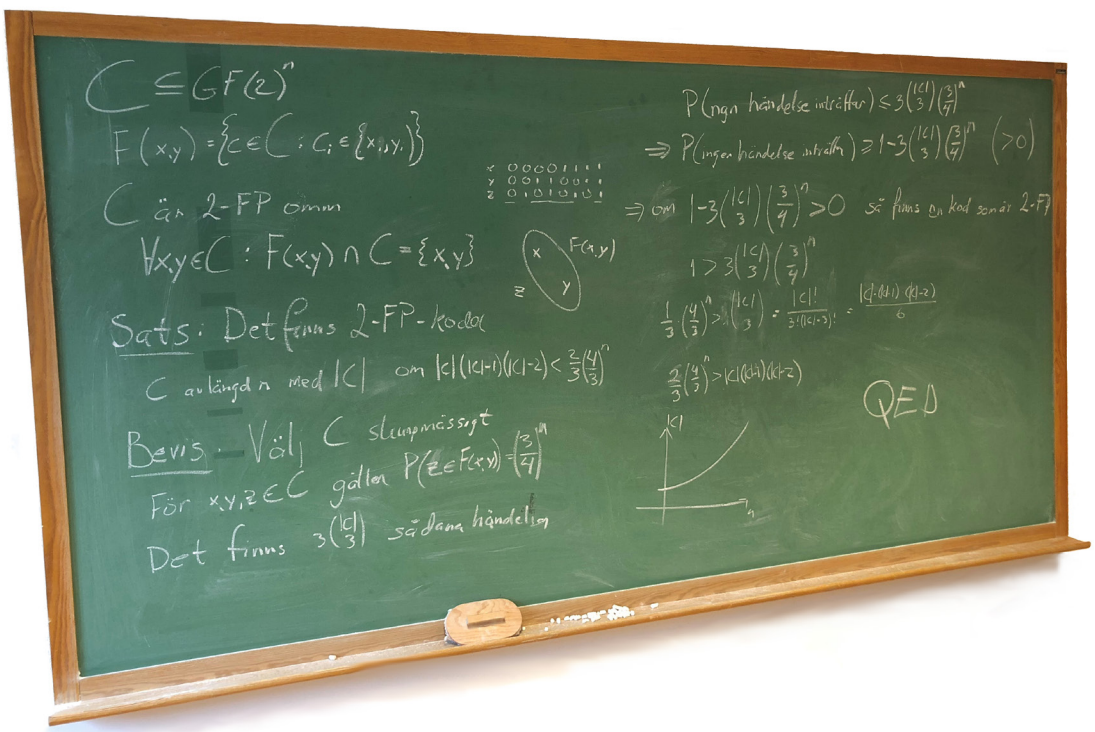
Praktisk mjukvaruverifiering med formella metoder

Ett hjälpmedel i granskningsprocessen

CAROLINE BILDSTEN, MIKA COHEN,
DANIEL EIDENSKOG, IOANA RODHE



FOI-R--4642--SE
ISSN 1650-1942 Oktober 2018

Caroline Bildsten, Mika Cohen,
Daniel Eidenskog, Ioana Rodhe

Praktisk mjukvaruverifiering med formella metoder

Ett hjälpmedel i granskningsprocessen

Titel	Praktisk mjukvaruverifiering med formella metoder − Ett hjälpmedel i granskningsprocessen
Title	Formal verification of software in practice − An aid in the review process
Rapportnr/Report no	FOI-R--4642--SE
Månad/Month	Oktober
Utgivningsår/Year	2018
Antal sidor/Pages	49
ISSN	1650-1942
Kund/Customer	Försvarsmakten
Forskningsområde	4. Informationssäkerhet och kommunikation
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E72727
Godkänd av/Approved by	Christian Jönsson
Ansvarig avdelning	Ledningssystem
Exportkontroll	Innehållet är granskat och omfattar ingen information som är underställd exportkontrollagstiftningen.

Sammanfattning

Försvarsmaktens IT-system har ofta höga säkerhetskrav och det behövs därför hög tilltro till att systemen fungerar enligt specifikation. För att uppnå hög tilltro krävs ett noggrant verifieringsarbete.

Verifiering av mjukvara är ett komplext problem där många olika metoder och verktyg måste samverka för att nå en hög tilltro till att mjukvaran är korrekt. Att mjukvaran är korrekt avseende både grundfunktionalitet och IT-säkerhets-funktioner är avgörande när det kan påverka exempelvis personsäkerhet eller skyddsvärd information. För att med hög tilltro kunna uttala sig om mjukvarans kvalitet krävs i regel ett omfattande arbete där såväl specifikation som implementation måste undersökas på djupet.

Denna rapport presenterar en studie som har undersökt hur verktyg som nyttjar formella metoder kan användas i praktiken vid verifiering av källkod. Arbetet har centrerats kring att verifiera några utvalda säkerhetsegenskaper, det vill säga enskilda aspekter av IT-säkerhetsfunktioner, baserat på två säkerhetsmål på hög nivå. Verifieringen har gjorts på källkod skriven i programspråket C där källkoden hör till en säkerhetskomponent vars funktion skulle kunna passa i många av Försvarsmaktens IT-system.

Verktygen har en stor spännvidd i fråga om bevisförmåga, dvs. styrkan hos de resultat som produceras av verktygen. Bland de verktyg som har använts i denna studie syns även en koppling mellan utsagornas styrka och hur komplicerat det är att använda verktygen, där starkare bevis krävde större arbetsinsats. Verktygen som använts i studien nyttjar formella metoder som bedömdes lämpliga för verifieringsproblemet, är mogna och tillgängliga för användning samt är kopplade till supportfunktioner eller aktiva användarforum.

Studiens slutsats är att användning av verktyg för verifiering med formella metoder på ett effektivt sätt kan ge ökad tilltro vid verifiering av säkerhetskritisk mjukvara. Det finns dock svårigheter med att använda sådana metoder enbart i granskningsfasen. Metoderna bör användas redan i utvecklingsstadiet när krav och kod kan anpassas. För att nå god effekt krävs lämplig metodik och kunnig personal. Såväl arbetsinsatsen som svårighetsgraden i användningen av verktygen anses vara hanterbara under dessa förutsättningar.

Nyckelord: formella metoder, abstrakt interpretering, symbolisk exekvering, Frama-C, CodeSonar, CBMC, KLEE

Summary

The IT systems of the Swedish Armed Forces often have strong security requirements and there is thus a need for a high level of confidence that the systems work according to specification. To acquire a high level of confidence a thorough verification is needed.

Software verification is a complex problem where many different methods and tools need to be used in concert in order to gain a high level of confidence in that the software is correct. Software correctness, regarding both functionality and IT security functions, is crucial when it concerns aspects like personal safety and protection of sensitive information. An extensive effort is usually required to acquire confidence in the quality of the software, especially since both specification and implementation must be examined in depth.

This report presents a study that investigates how verification of source code can be carried out practically, using tools that employ formal methods. The work has focused on verifying some selected security features, i.e. individual aspects of IT-security functionality, based on two high-level security goals. The study used source code written in the C programming language that constitute a security component whose functions could fit many IT systems in the Swedish Armed Forces.

The tools have a wide range of proof ability, which naturally reflects in the strength of the results produced by the tools. The proof ability also correlates with the complexity of using the tools, where stronger results require more effort from the user. The tools selected were ones that include formal methods considered appropriate for the verification problem, were mature and readily available, and offered support or an active user forum.

The study's conclusion is that, using appropriate methodology and knowledgeable personnel, the use of tools employing formal methods can give increased confidence in verification of security-critical software. Formal methods are not appropriate to use solely in the audit phase, in which it can become time-consuming to reverse engineer requirements and rewrite code. To fully appreciate formal methods as a technique for verification, the method should be incorporated in the development process to write formal requirements and verifiable code. In conclusion, we consider the workload and the difficulty level of using the tools as manageable under these circumstances.

Keywords: formal methods, abstract interpretation, symbolic execution, Frama-C, CodeSonar, CBMC, KLEE

Innehåll

1 Introduktion 6
1.1 Syfte och mål 7
2 Bakgrund 8
2.1 Formella krav 8
2.2 Resonemang 9
2.3 Abstraktion 11
2.4 Begränsningar i formella metoder 12
2.5 Arbetsinsats 13
2.6 Statisk analys 14
3 Metod 16
3.1 Definiera verifieringsproblem 16
3.2 Identifiera verktyg 16
3.3 Använd identifierade verktyg 17
3.4 Sammanställ erfarenheter 17
4 Verifieringsproblem 18
4.1 Säkerhetsmål 18
4.2 Säkerhetsegenskaper 19
5 Verktyg och erfarenheter 23
5.1 Urval och motivering 23
5.2 Installation 23
5.3 Frama-C 24
5.4 CBMC 31
5.5 KLEE 36
5.6 CodeSonar 37
6 Diskussion 41
7 Resultat och slutsatser 44
7.1 Rekommendationer 44
7.2 Sammanfattning av resultat 45
Referenser 48

1 Introduktion

Försvarsmakten har IT-system som behandlar uppgifter med hög sekretess eller har höga tillgänglighetskrav, vilket kräver hög tilltro till systemen. Detta är arbetskrävande att uppnå, vilket motiverar utvärderandet av nya metoder som kan bidra till byggandet av tilltro för att stödja det nödvändiga verifierings-arbetet. I detta arbete har praktisk användning av formella metoder studerats för att bedöma potentiella effekter av ett sådant införande och resultatet syftar bland annat till att informera MUST SÄKK om vilken förmåga som erbjuds av publikt tillgängliga verktyg. Studien är genomförd inom ramen för FoT-projektet *IT-säkerhetsmetoder* som är en del av FoT-området *Operationer i cyberdomänen.*

Idag har IT-system centrala uppgifter i många verksamheter, såväl inom Försvarsmakten som hos andra myndigheter och företag. IT-system kan utgöra delar av kärnverksamheten eller stödja viktiga funktioner som möjliggör eller underlättar arbetet. För att nå effekt förlitar sig verksamheterna på att IT-systemen är pålitliga och tillgängliga för att stödja den önskade operativa förmågan.

IT-systemen är inte enbart positiva för verksamheterna, utan för även med sig nya risker. Riskerna kan beröra många olika aspekter där negativa händelser kan påverka många olika tillgångar, exempelvis informationen som hanteras i systemen, maskiner och anläggningar som systemen styr samt systemen i sig. Negativa händelser som påverkar systemen kan även indirekt innebära risker för personer, exempelvis om en maskin inte beter sig korrekt, om kritiska beslut tas baserat på information som är felaktig eller om någon använder läckt information för att åsamka skada. För att reducera sannolikheten för oönskade händelser till en acceptabel nivå behöver en tillräckligt hög säkerhetsnivå upprätthållas för de kritiska delarna av systemen.

Viktiga delar av säkerhetsarbetet vid utveckling av IT-system är att troliggöra att säkerhetsrelevant mjukvara i systemet fungerar korrekt och att minimera antalet säkerhetsbrister. För att uppnå hög tilltro till systemets egenskaper behöver hela utvecklingskedjan beaktas, från verksamhetsanalys till färdig produkt och med hänsyn till alla relevanta leverantörer. Detta kräver en sund utvecklingsmetodik där bland annat entydiga och verifierbara krav definieras och valideras, programmering sker enligt kodstandarder och funktions- och säkerhetstester genomförs. För kritiska delar av mjukvaran kan det finnas anledning att använda sig av kodgranskning och formella metoder för att nå ännu högre tilltro till systemet.

Med *formella metoder* går det matematiskt att bevisa olika egenskaper hos mjuk-varan, exempelvis att den är korrekt utifrån en formell specifikation. Vidare är det möjligt att utifrån generella regler detektera olika typer av säkerhetsbrister, exempelvis buffer overflows. På grund av storleken på den arbetsinsats som

krävs använder formella metoder i alla praktiska fall verktygsstöd i form av programvara.

1.1 Syfte och mål

Syftet med studien som presenteras i denna rapport var att undersöka om tillgängliga verktyg för formella metoder är användbara för att verifiera säkerhetskritisk mjukvara i Försvarsmaktens IT-system.

Målet var att behandla ett realistiskt verifieringsproblem med relevanta verktyg och samla erfarenheter av detta. I detta ingick att utvärdera verktyg för formella metoder på ett utvalt testobjekt. I utvärderingen skulle olika aspekter på såväl bevisförmåga som användbarhet tas upp.

För att uppnå målet identifierades följande frågeställningar som beaktades för respektive verktyg. Frågorna tog upp olika aspekter som var av intresse i undersökningen för att förstå hur verktygen kunde nyttjas och för att identifiera deras begränsningar.

- Finns det verktyg som kan användas för att verifiera (delar av) säkerhetsmålen i studien?
- Vilken bevistyp ger verktygen?
- Hur användbart är verktyget?
    - Vilken är mognadsgraden hos verktygen med avseende på installation och support?
    - Hur komplext är det att använda verktygen?
    - Vilken kompetens krävs för att använda verktygen?

2 Bakgrund

Att testa mjukvara är notoriskt svårt, ofta betydligt svårare än att testa fysiska system. Utifrån en begränsad mängd testfall kan vi utan större problem ofta uttyda hur ett (kontinuerligt) fysiskt system beter sig generellt, men efter att en mjukvara passerat en testsvit är det svårt att säga hur mjukvaran beter sig i de fall som inte ingick i testsviten. Betrakta funktionen:

```
f(i: int)
  return … i / (3141-i) …
```

Så länge testsviten inte inkluderar ett specifikt indata, *i = 3141*, missar testningen att funktionen kan krascha (p.g.a. division med 0); mjukvaran beter sig problemfritt för alla indata utom ett fall. Testning vägleds och kompletteras därför ofta med kodgranskning, ett försök att mer eller mindre noggrant och systematiskt inspektera programkoden och resonera om möjliga programexekveringar. Kodgranskning är dock oftast tidskrävande, ansträngande och monotont, vilket gör kodgranskning både kostsam och opålitlig.

Formella metoder är ett försök att göra kodgranskningen rigorös och, i varierande mån, automatiserad. Formella metoder resonerar i princip om alla möjliga programexekveringar – programmet blir *bevisbart korrekt*.

Värdet av en mer rigorös kodgranskning blir särskilt tydligt vid säkerhetstestning som ska säkerställa att en mjukvara kan stå emot kreativa angripare, vilka inte begränsar sig till slumpmässigt felaktiga indata utan kan välja vilken indata som helst för att utnyttja en säkerhetsbrist.

Detta kapitel ger en kortfattad introduktion till formella metoder. Dessvärre använder författare inom formella metoder olika definitioner för vissa grundläggande begrepp och har skilda indelningar av de olika typerna av formella metoder. Därför vill vi göra läsaren uppmärksam på att det kan finnas ett behov av att jämka samman definitionerna i detta kapitel med definitioner som återfinns i andra texter.

2.1 Formella krav

Ett program är inte korrekt eller inkorrekt i sig utan är så endast relativt krav på programmet. När en formell metod verifierar ett program är det alltid ett antal specifika krav som verifieras, inte korrekthet i största allmänhet.

För att ett krav ska kunna verifieras måste det först formaliseras, vilket vanligen sker genom att lägga in assert-satser i programkoden, satser som hävdar att ett visst villkor är uppfyllt. Exempelvis kan kravet på frånvaro av division med 0 i

Kodexempel 1 formaliseras med en assert-sats `assert (3141-i) != 0` omedelbart innan divisionsinstruktionen.

Generiska exekveringsfel som är vanliga i det aktuella programspråket, exempelvis division med 0 i C, brukar verktyg för formell verifiering fånga genom att lägga in erforderliga assert-satser på egen hand. Funktionella krav måste däremot i allmänhet formaliseras av användaren själv. Detta sker då ofta i form av kodkontrakt (Meyer, 1997) (Tillmann & Schulte, 2005) som specificerar att en programfunktion garanterar ett visst villkor på resultatet givet ett visst (annat) villkor på indata. Villkoret på resultatet som funktionen ska garantera uttrycks härvid med en assert-sats, medan det funktionen får anta om indata ofta uttrycks med en assume-sats[1], en form av assert-sats som ber verifieringsverktyget att anta att ett visst villkor är uppfyllt.

Det är viktigt att komma ihåg att formella metoder inte bevisar överensstämmelse med högnivåkrav, så som vi människor intuitivt förstår dem, utan bevisar mot formella specifikationer som i sin tur har en mer eller mindre uppenbar koppling till högnivåkrav. Att omvandla informella, intuitiva och abstrakta högnivåkrav (exempelvis sekretess och integritet) till formella specifikationer är en svår konst.

2.2 Resonemang

Formella metoder är för många närmast synonymt med *resonemang*. En korrekthetsutsaga, en utsaga om att ett program uppfyller ett krav, verifieras genom att utsagan översätts till en logisk formel som avgörs med hjälp av en teorembevisare.

Verifikationsproblemet är dessvärre i allmänhet oavgörbart. En teorembevisare kan inte på egen hand avgöra om ett godtyckligt program uppfyller en (icke-trivial) egenskap. Inom *deduktiv verifiering* accepteras oavgörbarhet och resonemanget förväntas ske med stöd av mänsklig intelligens. Eftersom en användare måste ta del av och engageras i deduktionen används oftast speciella logiker, s.k. *programlogiker*, som är tänkta att erbjuda programmerare ett naturligt sätt att härleda assert-satser. En programlogiks slutledningsregler beskriver hur vi kan härleda nya assert-satser i programkoden utifrån programmets struktur och redan härledda assert-satser. En slutledningsregel för if-satser skulle t.ex. kunna säga att om vi redan har härlett en assert-sats på raden ovanför en if-sats, dvs. via en serie härledningar har vi fått ett program med strukturen:

[1] Assume sv: anta. Assume-satsen avbryter exekveringen om dess villkor inte är uppfyllt istället för att (som en vanlig assert-sats) kasta en exception. Assume-satser är ett språktillägg som verifieringsverktyg brukar erbjuda för att kodkontrakt och liknande specifikationer ska kunna uttryckas mer koncist.

```
...
assert <condition1>
if <condition2> then
  ...
```

kan vi lägga till `assert <condition1> and <condition2>` på första raden inuti if-satsen:

```
...
assert <condition1>
if <condition2> then
  assert <condition1> and <condition2>
    ...
```

under antagandet att villkoren `<condition1>` och `<condition2>` saknar sidoeffekter. Om vi redan visat att det förra programmet är giltigt, dvs. dess assert-satser är sanna oavsett indata, kan vi sluta oss till att även det senare programmet är giltigt. Med slutledningsregler i denna stil kan assert-satser propageras genom programmet, exempelvis från en inledande assume-sats i en funktionskropp och ner till en assert-sats för returvärden, för att på så sätt bevisa ett kodkontrakt för funktionen.

Formella metoder handlade länge framförallt om deduktiv verifiering. Sedan mer än ett decennium tillbaka har fokus dock allt mer flyttat till *model checking*, vilket gör att deduktiv verifiering inte längre röner samma uppmärksamhet. Deduktiv verifiering har dock fortsatt att utvecklas och aspekter därifrån, så som resonemang, har införlivats och blivit en väsentlig komponent i model checking.

Inom model checking görs verifikationsproblemet avgörbart genom att anta en bestämd övre gräns på resurserna som programmet (som ska verifieras) kan förbruka, exempelvis en övre gräns för storleken på indata eller det maximala antalet instruktioner (programsteg) som kan följa på varandra. De övre gränserna gör tillståndsrymden ändlig och verifikationsproblemet därmed avgörbart.

Symbolisk exekvering är förmodligen den mest populära formen av model checking just nu. Symbolisk exekvering introducerades redan på 80-talet (King, 1976), men först under det senaste decenniet har tekniken väckt ett större intresse med en rad anslående tillämpningar. T.ex. användes nyligen symbolisk exekvering som en viktig byggsten i de tre bidrag som vann guld, silver och brons i DARPA Cyber Grand Challenge[2]. Symbolisk exekvering används dagligen internt inom Microsoft för att säkerhetstesta exempelvis Windows och

[2] https://www.darpa.mil/news-events/2016-08-05a [läst 2018-06-15]

Office (Godefroid, 2015). Dessutom är symbolisk exekvering numera integrerat i Microsoft Visual Studio[3] och i Microsoft Security Risk Detection[4].

Grundidén i symbolisk exekvering är enkel och naturlig. Den symboliska exekveringen kan ses som en slags emulator som kör programkoden med symboliska indata som representerar godtyckliga, och därmed alla, konkreta värden. Emulatorn utforskar möjliga vägar genom koden genom att grena emuleringen vid varje villkor (*if-then-else*) och skapa en emuleringsgren för vardera villkorsgren. I den första grenen antar emuleringen att villkoret är uppfyllt (dvs. symboliska indata är sådana att villkoret är uppfyllt), i den andra grenen antar emuleringen tvärtom att villkoret inte är uppfyllt. När emulatorn stöter på en assert-sats tillfrågas en automatisk teorembevisare om negationen av assert-satsen och de antaganden som lett fram till den aktuella grenen samtidigt kan vara uppfyllda. Om så är fallet genererar teorembevisaren konkreta indata som leder till det felaktiga tillståndet.

Symbolisk exekvering utgör ett sätt att kompilera program till logiska formler. Även i andra former av model checking, såsom *symbolisk model checking* och *bounded model checking*, verifieras att ett program har en viss egenskap (exempelvis frånvaro av division med 0). Detta görs genom att programmet kompileras till logiska formler som beskriver alla tillstånd som programmet kan hamna i, varpå en teorembevisare används för att avgöra om logiska formlerna implicerar egenskapen ifråga, dvs. om egenskapen uppfylls i alla möjliga programtillstånd.

Resonemanget sker helt automatiskt med model checking. Programmeraren ger verktyget sin programkod och ges svar i form av problematiska indata, det vill säga indata som leder till att en assert-sats bryts, eller ett besked om att inga sådana indata existerar. Priset för automatiseringen är, som tidigare påpekats, att resonemanget antar en bestämd övre gräns på resurserna som programmet (som ska verifieras) kan förbruka, såsom en övre gräns för storleken på indata.

2.3 Abstraktion

Abstraktion utgör en annan grundläggande formell metod för att automatisera kodgranskning. Abstraktion innebär att ett program exekveras med abstrakta, förenklade datatyper på ett sätt som gör det möjligt att ur ett litet antal exekveringar sluta sig till hur programmet beter sig i allmänhet, för godtyckliga indata.

Vi illustrerar med en funktion för att beräkna absolutbeloppet av ett heltal:

```
abs(i: int)
```

[3] https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/introduction [läst 2018-06-15]
[4] https://www.microsoft.com/en-us/security-risk-detection/ [läst 2018-06-15]

```
    if i >= 0 then return i else return -1 * i
```

Säg att vi vill försäkra oss om att funktionen alltid returnerar ett icke-negativt heltal utan att behöva pröva varje möjlig indata *i*. Om vi förenklar programmet och ersätter datatypen *int* med en abstrakt datatyp {*neg*, *0*, *pos*}, i vilken *neg* och *pos* representerar negativa respektive positiva heltal,

```
abs(i: {neg,0,pos})
  if i >= 0 then return i else return neg * i
```

får vi ett så pass förenklat program att vi utan vidare kan testa alla indata (*neg*, *0* och *pos*). Om det förenklade programmet aldrig returnerar *neg*, kan vi sluta oss till att det ursprungliga programmet (med godtyckliga heltal som indata) aldrig returnerar ett negativt heltal.

Slutledningen (från ett förenklat program till det ursprungliga programmet) förutsätter att den abstrakta datatypen speglar den konkreta datatypen som ersätts. För abstraktionen ovan betyder det att operationerna * och > på den abstrakta datatypen måste spegla * och > på den konkreta datatypen *int*:

*pos * pos = pos*, *pos * neg = neg*, *0 * pos = 0*, *pos > 0*, etc.

Huruvida just dessa definitioner verkligen speglar multiplikation m.m. på datatypen *int* beror förstås på hur heltalsaritmetik har implementerats i det aktuella programspråket.[5]

Abstraktion introducerades som en formell metod med *abstrakt interpretering* (Cousot & Cousot, 1997) som idag har hunnit få ett flertal kommersiella implementationer. Historiskt har abstrakt interpretering framförallt använts för att utveckla säkerhetskritiska, inbäddade system, men under senare år har abstrakt interpretering i viss mån även spridit sig till mer konventionell mjukvaruutveckling hos t.ex. Facebook[6]. Abstraktion har också med tiden blivit en del av andra formella metoder, inte minst model checking.

2.4 Begränsningar i formella metoder

Formella metoder utlovar *bevisbart korrekta* program, men i det finstilta finns alltid mer eller mindre långtgående reservationer. Även inom deduktiv verifiering, den mest ambitiösa formen av formella metoder, måste bevis tas med viss försiktighet. För att underlätta verifieringen görs olika antaganden och förenklingar kring programsemantiken och det omgivande systemet, även

[5] Definitionerna speglar inte konkret multiplikation på ett korrekt sätt i exempelvis programspråk med heltalsöverflöde (integer overflow).
[6] http://fbinfer.com/ [läst 2018-06-15]

oriktiga sådana (såsom att datatypen *int* i C-kod beter sig som matematiska heltal).

Model checking lämnar garantier med en långtgående reservation: programmet är bevisbart korrekt givet en bestämd övre gräns på indata och andra resurser som programmet förbrukar. Hur programmet beter sig på exempelvis större indata säger verifieringen ingenting om. Å andra sidan tenderar model checking i mindre utsträckning än deduktiv verifiering att göra förenklande antaganden kring programsemantiken och det omgivande systemet.

Abstrakt interpretering ger i teorin bevisbart korrekta program. I praktiken lämnar verktygen sällan några garantier alls. Abstrakt interpretering producerar en stor mängd falsklarm, dvs. varningar ges trots att programkoden inte innehåller något fel. För att få ner mängden falsklarm till en mer acceptabel nivå filtrerar nästan alla verktyg bort larm enligt vissa heuristiker och även korrekta larm riskerar då att fastna i filtren.

Det kan vara lätt att glömma bort att formell verifiering bevisar kodens korrekthet relativt en formell specifikation (såsom assert-satser) och inte relativt informella, intuitiva egenskaper. Verktyg som tillåter användare att formalisera egenskaper på en högre, mer intuitiv och naturlig abstraktionsnivå skalar än så länge inte riktigt till konkret källkod (Lomuscio, Qu, & Raimondi, 2017).

2.5 Arbetsinsats

Formella metoder är automatiska i olika grad. Det bör dock understrykas att en formell metod brukar räknas som automatiserad i den mån den formella, matematiska analysen är automatiserad och användaren själv slipper genomföra de matematiska handgreppen. Att en formell metod är *automatisk* är alltså i sig ingen garanti för att den inte kräver åtskilligt annat manuellt arbete.

Abstrakt interpretering räknas som en automatisk metod. Det är i stort sett bara att ge källkoden till analysverktyget och invänta svar, vilket sällan tar lång tid eftersom abstrakt interpretering inte kräver mycket beräkningsresurser, inte ens för stora mängder kod. Dock genererar abstrakt interpretering rikligt med falsklarm, och varje larm måste noggrant undersökas innan det kan avfärdas som falskt.

Även model checking räknas som en automatisk metod. Programmeraren ger programkoden till verktyget och får antingen ett problematiskt testfall, det vill säga indata som utlöser en bugg, eller ett besked om att inga sådana indata existerar. I praktiken kräver dock model checking en hel del extra programmering för att tillämpas. Beräkningskostnaden växer så snabbt med storleken på programkoden att denna oftast måste brytas ned i mindre, mer hanterbara programenheter som sedan verifieras var för sig så som inom enhetstestning. Idealt sammanfaller denna nedbrytning med den nedbrytning i mindre enheter

(funktioner, klasser, etc.) som ändå sker som del av systemutvecklingen. Med massiv beräkningskapacitet verkar model checking i vissa fall även kunna analysera större programkod, som i fallet med Microsofts rutinmässiga analys av källkod i ett dedikerat beräkningskluster (Godefroid, 2015).

Deduktiv verifiering, slutligen, räknas som en semi-automatisk metod eftersom deduktionen behöver guidas av en användare.

Deduktiv verifiering och även model checking (men inte abstrakt interpretering) används inte bara för att fånga generiska exekveringsfel, såsom minnesfel i C-program. Dessa tekniker kan även verifiera programspecifika funktionella krav, vilket då kan kräva en del arbete för att formalisera specifikationerna. Medan verktygen på egen hand fångar generiska exekveringsfel, behöver användaren i allmänhet presentera funktionella krav för verktyget med t.ex. assert-satser. Funktionella krav som är allmängiltiga för program i ett visst ramverk, t.ex. krav som Windows ställer på drivrutiner, kan dock finnas fördefinierade i verktyg som är specialiserade för just det ramverket (se exempelvis Microsoft Static Driver Verifier[7]).

2.6 Statisk analys

Formella metoder kan ses som en form av *statisk analys*. Andra, mer lättviktiga former av statisk analys strävar inte efter bevisbart korrekt kod, vilket som vi sett nästan alltid i en eller annan form är den bakomliggande agendan i formella metoder, utan producerar istället varningar utifrån en mer ytlig syntaxkontroll. I den vanligaste formen av lättviktig statisk analys, *efterlevnadskontroll*, kontrolleras att koden efterlever en rekommenderad kodningspraxis, exempelvis MISRA C.

Kontamineringsanalys är en annan form av lättviktig statisk analys som under senare år blivit populär inom bredare kretsar. Språk som Perl och Ruby har inbyggt stöd för kontamineringsanalys medan andra språk som C, Java och Python har verktyg som stödjer kontamineringsanalys. Analysen spårar data-flödet i programkod för att bedöma om känslig indata (exempelvis ett lösenord) kan läcka ut i publik utdata (så som ett oskyddat minne). Om detta är fallet varnas användaren för ett potentiellt konfidentialitetsbrott. Omvänt kan data-flödet spåras för att bedöma om publik, opålitlig indata kan korrumpera data vars

[7] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier [läst 2018-06-15]

riktighet ska skyddas. I dessa fall varnas användaren för ett potentiellt riktighetsbrott. Kontamineringsanalysen följer endast vissa typiska former av explicit dataflöde och missar därmed mer subtila, implicita informationsflöden[8].

Ett antal välspridda kommersiella produkter (exempelvis CodeSonar[9], Coverity[10] och MathWorks Verification and Validation[11]) erbjuder användaren en integrerad verktygslåda med olika slags statiska kodanalyser. Ett och samma verktyg kan presentera formella metoder tillsammans med lättviktig statisk analys under den gemensamma beteckningen statisk analys.

[8] Kontamineringsanalysen följer endast explicita dataflöden av formen `public_bit = secret_bit + 1`, där den hemliga `secret_bit` flödar till den publika `public_bit`. Analysen ignorerar implicita flöden av formen `if secret_bit then public_bit = 1 else public_bit = 0`, där `secret_bit` flödar till `public_bit` implicit via kontrollflödet.

[9] https://www.grammatech.com/products/codesonar [läst 2018-06-15]

[10] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html [läst 2018-06-15]

[11] https://se.mathworks.com/help/sltest/verification-validation-and-test.html [läst 2018-06-15]

3 Metod

Studien innehöll följande steg:

1. *Definiera verifieringsproblem* i form av mjukvara, säkerhetsmål och säkerhetsrelevanta egenskaper att verifiera.
2. *Identifiera verktyg* som kan användas för att verifiera säkerhetsrelevanta egenskaper med hjälp av formella metoder.
3. *Använd identifierade verktyg* för att verifiera säkerhetsrelevanta egenskaper med hjälp av formella metoder.
4. *Sammanställ erfarenheter* från användningen av verktygen.

Merparten av arbetet låg i steg 3, då användningen av identifierade verktyg utgjorde den huvudsakliga informationsinsamling som låg till grund för de erfarenheter och den diskussion som utgör studiens resultat.

3.1 Definiera verifieringsproblem

Det första steget var att välja ett testobjekt i form av en mjukvara som utgjorde en representativ, säkerhetsrelaterad funktion i ett tänkt IT-system inom Försvarsmakten. Av sekretesskäl undveks system som används i skarp drift inom Försvarsmakten. Istället valdes att studera fritt tillgängliga mjukvaror med öppen källkod.

För det valda testobjektet ställdes några relevanta säkerhetsmål upp, som sedan succesivt bröts ned till verifierbara säkerhetsegenskaper.

3.2 Identifiera verktyg

Urvalet av verktyg byggde på projektgruppens befintliga kunskap, kompletterad med sökningar på internet. Urvalskriterier ställdes upp för att välja ut verktyg som var praktiskt användbara i situationer som liknar faktiska utvecklingsprojekt.

Förutom att lämpa sig för att verifiera identifierade säkerhetsegenskaper ställdes följande krav på de verktyg som skulle användas i studien:

1. Verktyget ska nyttja formella metoder enligt definitionerna i kapitel 2.
2. Verktyget ska vara tillgängligt för användning, exempelvis genom att det är fritt tillgängligt eller att det går att få en utvärderingslicens.
3. Verktyget ska vara moget, med innebörden att det ska finnas dokumentation och att verktyget är stabilt vid användning.

4. Verktyget ska undergå fortlöpande utveckling med ett aktivt användarforum eller möjlighet att erhålla support.

3.3 Använd identifierade verktyg

Verktygen som identifierades i steg två installerades och applicerade på test-objektet. Under arbetets gång noterades positiva och negativa erfarenheter av verktygen.

3.4 Sammanställ erfarenheter

Utifrån ovanstående steg sammanställdes de erfarenheter som samlats in under användandet av verktygen och som var relevanta för studiens frågeställningar.

4 Verifieringsproblem

OpenVPN och mbed TLS pekades ut av Försvarsmakten som ett relevant testobjekt. Mjukvarorna är skrivna i programspråket C och har ett gott rykte vad gäller säkerhet.

De valda mjukvarorna, *OpenVPN 2.4.3* och *mbed TLS 2.5.1*, implementerar tillsammans funktioner för virtuella privata nätverk (eng. virtual private network, VPN), där OpenVPN står för övergripande VPN-funktionalitet och mbed TLS står för kryptografiska funktioner.

För att begränsa komplexiteten i mjukvaran avgränsades testobjektet till VPN-tunnlar som nyttjar kryptosviten ECDHE-ECDSA-AES128-GCM-SHA256. Detta innebar att övriga kryptosviter deaktiverades.

Motiveringen till valet av OpenVPN och mbed TLS är att dessa tillsammans utgör en säkerhetskomponent vars funktion passar i många av Försvarsmaktens IT-system. Båda mjukvarorna ger intryck av att hålla god kodkvalitet vid en översiktlig genomgång. Vissa delar av implementationen i mbed TLS har även verifierats med formella metoder (Ye, o.a., 2017) (TrustInSoft, u.å.)[12].

De specifika versioner av mjukvarorna som har använts i studien valdes då dessa var de senaste tillgängliga vid projektets start.

4.1 Säkerhetsmål

Utifrån mjukvarans funktion identifierades några få tekniska säkerhetsmål som huvudsakligen bygger på funktioner som har implementerats i mjukvaran. Valet av säkerhetsmål är inte heller kritiskt, men bör ha en tydlig koppling till typiska säkerhetsmål i Försvarsmaktens IT-system. I denna studie är säkerhetsmålen endast till för att ge konkreta uppgifter i det verifieringsarbete där verktygen används och behöver därmed inte vara heltäckande. Säkerhetsmålen som valdes utgör endast en liten delmängd av alla säkerhetsmål som vore aktuella om en komplett granskning av mjukvaran skulle göras.

Efter val av säkerhetsmål skaffades en övergripande förståelse för design och implementation av mjukvaran utifrån de valda säkerhetsmålen. Detta utfördes i form av genomgångar av källkoden, där främst kod som implementerar eller berörs av säkerhetsmålen undersöktes.

Slutligen bröts säkerhetsmålen ner till *säkerhetsegenskaper* i källkoden. Säkerhetsegenskaperna utgör konkreta och avgränsade målsättningar för en specifik

[12] mbed TLS hette tidigare PolarSSL. TrustInSoft verifierade en version som släpptes innan namnbytet.

funktion eller del av källkoden. Säkerhetsegenskaper kan exempelvis vara att en minnesyta skrivs över, att en korrekt beräkning utförs eller att ett korrekt beslut tas.

4.1.1 Säkerhetsmål 1 – Radering av nycklar

Systemet ska aktivt radera icke-publika kryptonycklar och data som används för att derivera icke-publika kryptonycklar när dessa inte längre behövs.

I säkerhetsmålet ingår både utförande av raderingen och beslut att radering ska utföras. Data som avses är nyckelmassa samt data som kan nyttjas för att förenkla återskapandet av hela nycklar eller delar av nycklar. Raderingen ska vara aktiv, vilket betyder att raderingen ska ske genom överskrivning av de minnespositioner där nycklar eller data har lagrats. Att deallokera minne, radera pekare eller liknande är inte en aktiv radering och därför inte en godtagbar metod.

4.1.2 Säkerhetsmål 2 – Kodkvalitet

Implementationen ska inte innehålla farliga kodkonstruktioner eller vanliga programmeringsfel.

Säkerhetsmålet utgör ett allmänt kodkvalitetsmål för att säkerställa en god lägstanivå på källkodens kvalitet. Målet specificerar inte explicita regler men lämpliga regeluppsättningar återfinns i exempelvis MISRA C:2012 (MISRA, 2012) och SEI CERT C Coding Standard (SEI CERT, 2016).

4.2 Säkerhetsegenskaper

I detta avsnitt beskrivs en partiell nedbrytning av säkerhetsmålen till konkreta säkerhetsegenskaper som kan användas vid verifiering av källkoden.

De två säkerhetsmål som beskrivs ovan har olika karaktär. Det första säkerhetsmålet är relativt konkret och berör förhoppningsvis bara en avgränsad del av källkoden. Det andra målet är betydligt mer abstrakt och berör källkoden som helhet.

Detta avsnitt avser inte att påvisa en komplett nedbrytning till säkerhetsegenskaper. Ett sådant arbete är omfattande och kräver mer tid för att tillse att viktiga egenskaper inte förbises. I den här studien har nedbrytningen förenklats och koncentrerats på att bryta ut ett fåtal säkerhetsegenskaper i syfte att ha realistiska exempel på egenskaper att testa.

4.2.1 Egenskaper för säkerhetsmål 1

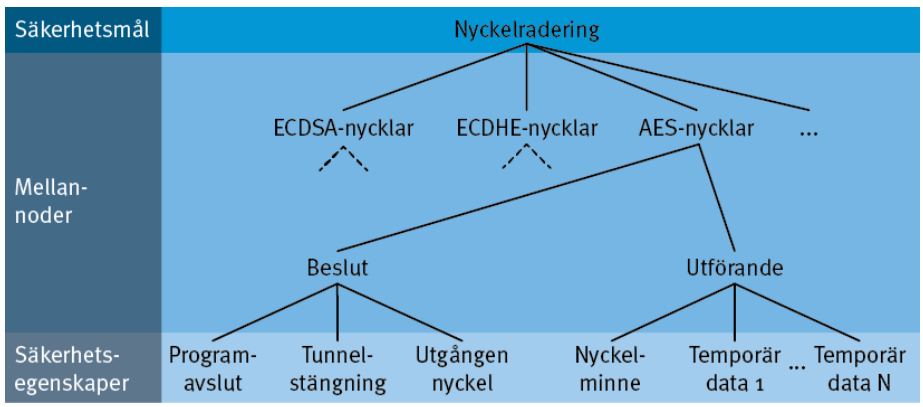
Säkerhetsmål 1 innebär att icke-publika nycklar ska raderas när dessa ej längre behövs. Detta mål berör många delar av källkoden då mjukvaran använder kryptonycklar för flera olika algoritmer vid ett antal olika tillfällen i funktionskedjan. För att hitta alla ställen i källkoden som påverkar radering och dess utförande måste säkerhetsegenskaper definieras för:

1. Allt minne där icke-publika nycklar lagras.
2. Allt minne där delar av nycklar och mellanresultat lagras vid användning och beräkning av icke-publika nycklar.
3. Alla punkter där såväl explicita som implicita beslut om radering tas.

Säkerhetsegenskaperna är inte entydigt definierade enbart utifrån säkerhetsmålet, utan är även beroende på design och implementation av mjukvaran. För att identifiera säkerhetsegenskaperna har därför hänsyn tagits till mjukvarans uppbyggnad och hantering av nyckeldata.

Då det inte är aktuellt med en komplett verifiering i denna studie har nedbrytningen till säkerhetsegenskaper begränsats till att gälla radering av de AES-nycklar som används för kryptering av trafik i VPN-tunneln. Figur 1 visar en delmängd av nedbrytningen till säkerhetsegenskaper, med fokus på AES-nycklarna.

När det gäller att identifiera de ställen i källkoden som berörs av säkerhetsegenskaperna är vissa lättare att identifiera än andra. Exempelvis är det i regel ganska rättframt att identifiera var en nyckel lagras mellan användningar, medan det är betydligt mer komplext att identifiera alla ställen där temporär data hanteras i form av mellanresultat i beräkningar.



Figur 1. Partiell nedbrytning av säkerhetsmål 1 till egenskaper.

Figur 1 visar hur nedbrytningen av säkerhetsmål bygger upp ett träd där löven utgör säkerhetsegenskaperna. För att nå fullständig verifieringstäckning behöver varje säkerhetsmål sannolikt brytas ner till ett stort antal säkerhetsegenskaper, som var och en kräver olika verifieringsmetoder där formella metoder kan utgöra en delmängd.

Följande lista tar upp några säkerhetsegenskaper som har identifierats för säkerhetsmålet, avgränsat till att enbart gälla AES-nycklar, inklusive referenser till relevanta funktioner i källkoden.

- **Raderingsbeslut vid stängning av VPN-tunnel**
  tunnel_point_to_point (openvpn/src/openvpn/openvpn.c:61)
  close_instance (openvpn/src/openvpn/init.c:4182)
- **Raderingsbeslut för utgångna nycklar**
  tls_multi_process (openvpn/src/openvpn/ssl.c:3124)
  lame_duck_must_die (openvpn/src/openvpn/ssl.c:1215)
- **Radering av nyckelminne**
  tls_session_free (openvpn/src/openvpn/ssl.c:1127)
  key_state_free (openvpn/src/openvpn/ssl.c:970)
  key_state_ssl_free (openvpn/src/openvpn/ssl_mbedtls.c:1004)
  mbedtls_cipher_free (openvpn/src/openvpn/cipher.c:138)
  mbedtls_ssl_free (mbedtls/library/ssl_tls.c:7341)
  mbedtls_zeroize (mbedtls/library/cipher.c:66)
- **Raderingsbeslut vid programavslut som följd av felhantering**
  Denna egenskap är utspridd i koden då felhantering utförs på många ställen i mjukvaran.
- **Radering av temporär data som kan användas för att återskapa nyckeldata.**
  Denna egenskap saknar tydlig kodreferens då egenskapen är spridd i källkoden. För denna egenskap kan det vara fördelaktigt med verktygsstöd för att identifiera vilka variabler och minnesytor som berörs av nyckeldata.

Verifiering av den här typen av konkreta säkerhetsmål kan underlättas avsevärt om säkerhetsmålen har spårats genom design och implementation ner till säkerhetsegenskaperna, exempelvis genom annotering i dokumentation och källkod.

4.2.2 Egenskaper för säkerhetsmål 2

Säkerhetsmål 2, som berör generell kodkvalitet, kan brytas ner efter källkodens struktur, men det blir snabbt otympligt då det kan leda till att många likvärdiga

säkerhetsegenskaper tas fram för olika stycken av källkod. För detta säkerhetsmål är det lämpligare att basera säkerhetsegenskaperna på programspråkets egenskaper och de vanligt förekommande fel som görs i språket.

För att säkerställa att känd kunskap om farliga konstruktioner och vanliga programmeringsfel hanteras inom säkerhetsmålet väljs säkerhetsegenskaperna utifrån publicerade regler för säker programmering. För att hålla antalet säkerhetsegenskaper på en rimlig nivå i denna studie valdes en liten delmängd ut bland de regler som ingår i SEI CERT C Coding Standard (SEI CERT, 2016). De regler som valdes är:

- ARR30-C – Skapa inte felaktiga pekare eller index till arrayer.
- EXP33-C – Läs inte oinitierat minne.
- MEM30-C – Använd inte deallokerat minne.
- MEM31-C – Frigör dynamiskt allokerat minne som inte längre behövs.
- STR31-C – Garantera att null-tecken får plats i strängar.

5 Verktyg och erfarenheter

I detta kapitel introduceras och motiveras valet av de verktyg som inkluderades i studien samt erfarenheter från användningen av dessa.

5.1 Urval och motivering

Urvalet av verktyg följer de kriterier som beskrivs i avsnitt 3.2. Sammanfattningsvis är dessa kriterier att verktygen ska vara ändamålsenliga, nyttja formella metoder, vara tillgängliga för användning samt vara stabila och undergå fortlöpande utveckling. Det ska även finnas möjlighet till support eller finnas ett aktivt användarforum.

Eftersom verktygen ska vara tillgängliga för användning samt vara möjliga att få tag på, har i första hand fritt tillgängliga verktyg med öppen källkod valts ut. Alla verktyg bygger på någon form av formell metod eller kombination av formella metoder enligt beskrivningen i kapitel 2. De formella metoder som studien har fokuserat på är abstrakt interpretering, deduktiv verifiering och model checking.

När det gäller model checking finns det en rik flora av verktyg. CBMC och KLEE valdes då de är två av de ledande verktygen. För deduktiv verifiering och abstrakt interpretering valdes Frama-C med motiveringen att det är ett etablerat verktyg som delvis var bekant sedan tidigare.

CodeSonar, som är ett kommersiellt verktyg, valdes då det är ett väl etablerat verktyg med relativt många användare. CodeSonar har även funktionalitet för att testa ett programs kodkvalitet, något som passar bra in på säkerhetsmål 2. Eftersom det är ett kommersiellt verktyg så uppfyller det endast delvis tillgänglighetskriteriet, dock var det möjligt att testa verktyget genom en 30-dagars utvärderingslicens.

Samtliga verktyg som valdes är etablerade, undergår fortlöpande utveckling och har möjlighet till support.

5.2 Installation

Samtliga verktyg har installerats och körts på en Linux-maskin med OpenSuSE Tumbleweed, en Linux-distribution som snabbt tillhandahåller nya versioner av de mjukvaror som ingår i distributionen. Detta har inneburit en del problem för de verktyg som är öppen källkod, det vill säga Frama-C, CBMC och KLEE, då det fanns olika direkta eller indirekta beroenden mot äldre versioner av olika mjukvaror som inte längre fanns tillgängliga i den använda Linux-distributionen. De äldre versioner som krävdes av mjukvarorna var i regel så pass gamla att de

redan utgått ur samtliga Linux-distributioner, varför valet av Linux-distribution inte var avgörande.

De äldre versionerna av aktuell mjukvara var bitvis svårinstallerade och krävde goda kunskaper i Linux för att få på plats, något som kan ge problem vid installation i en produktionsmiljö. Detta kan bli speciellt påtagligt om Linux-miljön ska hållas uppdaterad, något som ofta är önskvärt vid kontinuerlig utveckling.

Installationsprocessen för CodeSonar var väldokumenterad och problemfri.

En ytterligare notering vid tidig provkörning av CBMC och CodeSonar var att dessa inte förstod den nya datatypen _Float128, vilket hindrade analys av mjukvara som använde funktioner från systemets standardbibliotek. Datatypen är relativt ny i standarden för C++ och infördes nyligen i den utvecklingsmiljö för C/C++ som används i flertalet Linux-distributioner, det vill säga kompilatorn *GCC* och biblioteket *glibc*.

Problemet med _Float128 gick att kringgå med direktiv till kompilatorn, även om det innebär en viss risk att vissa fel i källkoden inte upptäcks. I de aktuella mjukvarorna används dock inte _Float128, varför det torde vara riskfritt i dessa fall. Nyare versioner av verktygen kommer rimligtvis att åtgärda detta problem.

5.3 Frama-C

Frama-C[13] står för ”Framework for Modular Analysis of C code” – ramverk för modulär analys av C-kod.

Det finns flera insticksmoduler till verktyget som möjliggör olika former av analys, både statiska analysmetoder (i form av abstrakt interpretering och deduktiv verifiering) och dynamiska analysmetoder. Frama-C är utvecklat av franska CEA[14] och INRA[15]. Det är baserat på öppen källkod och stöds av en community som har utvecklat flertalet tredjepartsmoduler (Kirchner, Kosmatov, Prevosto, Signoles, & Yakobowski, 2015). De moduler som användes i studien var *Weakest Precondition* (WP) som utför deduktiv verifiering och *Evolved Value Analysis* (EVA) som utför abstrakt interpretering.

ANSI/ISO C Specification Language (ACSL) är det språk som används för att skriva formella specifikationer i form av kodkontrakt och annotationer till Frama-C. ACSL-annotationer används för att uttrycka funktionella egenskaper i källkoden. I kodkontrakten specificeras vad funktionen kräver av anropande funktion, så kallade *förvillkor*, och vad den garanterar för resultat, så kallade

[13] https://frama-c.com/ [läst 2017-12-28]
[14] Commissariat à l'Énergie Atomique et aux Énergies Alternatives
[15] Institut national de recherche en informatique et en automatique

eftervillkor. Frama-C använder teorembevisare för att framställa bevis för att kodkontrakten gäller och för att framställa motbevis om kontrakten inte uppfylls.

Frama-C kan skapa varningar för exekveringsfel (exempelvis otillåten minneshantering) om detta aktiveras i verktyget. Varningarna kan hanteras genom att korrigera koden eller genom att skriva kodkontrakt för att indikera för Frama-C att detta inte kan inträffa. Mer om denna funktion går att läsa i avsnitt 5.3.3.

5.3.1 Minst begränsande förvillkor

Minst begränsande förvillkor (eng. weakest precondition, WP) är en teknik för deduktiv verifiering som är baserad på arbete av Hoare, Floyd och Dijkstra (Kirchner, Kosmatov, Prevosto, Signoles, & Yakobowski, 2015) och som används i insticksmodulen WP. Modulen möjliggör formell verifiering av att implementation stämmer mot den formella specifikationen som har skrivits i språket ACSL.

WP-modulen innehåller två olika minnesmodeller, *Hoare* och *Typed* (Baudin, Bobot, Correnson, & Dargaye, 2016), där Typed väljs om inget annat anges. Hoare-modellen är begränsad, men lämplig då pekare inte är explicit tilldelade[16]. Typed-modellen är lämplig vid användning av deklarerade pekare. Ytterligare minnesmodeller är under framtagning till verktyget. De olika modellerna måste väljas beroende på fall, då det finns risk att verktyget inte kan producera något bevis om en olämplig modell används. I dessa fall måste modellen bytas eller så måste koden skrivas om för att passa till modellen.

WP-modulen behöver viss hjälp för att kunna verifiera ett kodkontrakt för en funktion. Bland annat krävs det speciella annotationer för alla loopar inuti funktionen. De annotationer som krävs för en loop är:

- *Loop invariant* – indikerar oföränderliga egenskaper i en loop som gäller både innan och efter varje iteration.
- *Loop assign* – indikerar vid starten av varje iteration vilka variabler som har förändrats i de iterationer som redan har körts.
- *Loop variant* – används för att bevisa att loopen avslutas. Den pekar ut en heltalsvariabel som minskar vid varje loop-iteration och som är begränsad till positiva heltal.

Resultatet av analysen i Frama-C visas grafiskt i användargränssnittet genom färgade ikoner. Tabell 1 innehåller de möjliga ikonerna för bevistillstånd för kodkontrakt och annotationer.

[16] Den har ingen representation av minnesutrymmet heap, vilket leder till att vissa typer av pekaranvändning inte alltid går att tolka.

Tabell 1. Översikt över ikoner för bevistillstånd.

	Inget försök att bevisa egenskapen har gjorts.
	Egenskapen har inte validerats.
	Egenskapen är giltig men har beroenden.
	Egenskapen och alla dess beroenden är giltiga.

5.3.2 Värdeanalys

Värdeanalys (”Evolved value analysis” i Frama-C) (Bühler, Cuoq, & Yakobowski, 2016) är en insticksmodul baserad på abstrakt interpretering (Kirchner, Kosmatov, Prevosto, Signoles, & Yakobowski, 2015). Värdeanalys kan användas för att hitta exekveringsfel samt som ett sätt att granska och förstå kod genom att i verktyget studera vad olika variabler kan anta för värden. Värdeanalys går även att kombinera med modulen WP för deduktiv verifiering av funktioner.

Insticksmodulen beräknar automatiskt möjliga värden för alla variabler i programmet som testas. När det finns flera olika exekveringsvägar så utforskar abstrakt interpretering varje möjlig väg.

Förutom att inventera vad variabler antar för värden så kan värdeanalys även detektera exekveringsfel. När den följer exekveringsvägarna under analys så kan verktyget detektera exempelvis division med noll och ogiltiga pekaranrop. Frama-C hävdar att de kan hitta alla exekveringsfel med sin metod. Eftersom verktyget använder värdeanalys kan verktyget dock även hitta falskt positiva exekveringsfel.

Det finns en inställning för att öka precisionen i analysen av exekveringsfel, vilket ger ett minskat antal falskt positiva resultat. Inställningen gör att programmet djupare analyserar loopar och villkorssatser, vilket dock leder till ökad analystid.

5.3.3 Annotationsgeneratorn RTE

Frama-C inkluderar en insticksmodul för generering av annotationer för vanliga exekveringsfel samt kontroll av kodkontraktens förvillkor och eftervillkor. Insticksmodulen är menad att användas som extrafunktion tillsammans med andra moduler, såsom minst begränsande förvillkor (WP) och värdeanalys. RTE utför själv ingen värdeanalys och kan därför inte detektera alla förekomster av exekveringsfel.

Följande typer av exekveringsfel kan detekteras med annotationer från RTE-modulen (Herrmann & Signoles, 2017):

- otillåten minnesanvändning
- division med noll
- odefinierade skiftoperationer
- oinitierade variabler och hängande pekare till lokala variabler
- odefinierade pekarjämförelser
- integer overflow
- sidoeffekter i uttryck
- otillåten funktionspekaraccess.

5.3.4 Skärning

Modulen för *skärning* (eng. slicing) returnerar ett program som är en delmängd av det program som analyseras. Detta kan vara en användbar funktion vid felsökning för att lättare hitta felkällor.

Delmängden baseras på ett *skärningskriterium* (Weiser, 1981). Extraherade programsatser och predikat formar en *skiva* (eng. slice).

Ett skärningskriterium definieras som $C = (x,V)$ där x är en sats i programmet och V är en delmängd av variablerna i programmet. Skivan kommer att inkludera alla satser som påverkar variablerna i V vid satsen x (Harman & Hierons, 2001).

Insticksmodulen använder resultatet från värdeanalysmodulen tillsammans med beräkningar av olika funktionsberoenden. Den stödjer skärningskriterier för att kunna göra kodobservationer eller för att kunna utföra olika bevis.

5.3.5 Användning av Frama-C

Vi har valt att fokusera användningen av Frama-C på WP-modulen, som använder tekniken *minst begränsande förvillkor*. Den möjliggör deduktiv verifiering på ett sätt som lämpar sig för att verifiera radering av nyckelminnet. Radering av nyckelminnet är en av de säkerhetsegenskaper som har identifierats för säkerhetsmål 1 i avsnitt 4.1.1. Det är möjligt att skärningsmodulen skulle kunna användas för att göra en kontamineringsanalys för att hitta hur nyckeln propageras i koden, men detta har inte testats.

```
//Function under test
static void mbedtls_zeroize(void *v, size_t n) {
       volatile unsigned char *p = (unsigned char*)v;
```

```
        while (n--){
                *p++ = 0;
        }

}
```

Figur 2. Funktionen *mbedtls_zeroize()* som ska testas.

```
/*@
  requires n > 0;
  requires \valid(v+(0..n-1));
  assigns v[0..n-1];
  ensures \forall integer q;
          0 <= q <= n-1 ==> v[q] == (unsigned char)0;
*/
static void mbedtls_zeroize(unsigned char *v, size_t n) {
        volatile unsigned char *p = (unsigned char*)v;

/@
  loop invariant 0 <= n <= \at(n, Pre);
  loop invariant p == v+(\at(n,Pre)-n);
  loop invariant \forall integer j;
         0 <= j < (\at(n, Pre)-n) ==> v[j] == (unsigned char)0;
  loop assigns n, p, v[0..\at(n, Pre)-n-1];
  loop variant n;
*/
        while (n--){
                *p++ = 0;
        }
}
```

Figur 3. Funktionen *mbedtls_zeroize()* med kodkontrakt och loop-annotationer skrivna i ACSL (omgivna av /*@ och */). Kodkontraktet anges ovanför funktionsdefinitionen och loop-annotationerna anges ovanför *while*-loopen.

Av de funktioner som berör denna säkerhetsegenskap väljer vi att titta närmare på *mbedtls_zeroize()* som visas i figur 2. Funktionen skriver över de minnespositioner som pekas ut av inparametrarna och Frama-C används för att verifiera att raderingen utförs korrekt av funktionen. Det vi vill bevisa är kodkontraktet i figur 3. Kontraktet innebär, givet att *v* är en giltig pekare och *n* ett positivt heltal, att funktionen skriver *n* nollor i minnet med start i position *v*. Om funktionen anropas med pekaren till en nyckellagring och nyckelns storlek som indata, vet vi därmed att nyckeln kommer att raderas som resultat av funktionsanropet (det vill säga att det skrivs nollor i minnet där nyckeln fanns).

Då WP-modulens interna minnesmodell bygger på statiska typdeklarationer stöds inte void-pekare (*void* *) särskilt bra. Därför har vi modifierat funktionen *mbedtls_zeroize()* så att variabeln *v* får typen *unsigned char* * istället för *void* *. Funktionellt sett är koden oförändrad, samtidigt som WP-modulen får den information den behöver. Utöver detta vill vi också påpeka att WP-modulen ignorerar nyckelordet volatile, vilket i vissa fall kan ha betydelse för resultatets giltighet.

Verktyget rapporterar att funktionen uppfyller sitt kodkontrakt, och vi sluter oss till att funktionen kommer att göra det som förväntas av den för alla möjliga giltiga indata.

Arbetsgång

Vidare följer en beskrivning av de annotationer som behövs i exemplet och även av arbetsgången. Till att börja med har kodkontrakt skrivits för funktionen. De olika raderna i kodkontraktet i figur 3 handlar om programtillståndet innan respektive efter funktionen har exekverats. Kodkontraktet för inparametrarna beskriver giltiga värden för dessa vid funktionsanropet:

```
requires n>0;

requires \valid(v+(0..n-1));
```

I vårt fall, där vi endast verifierar en funktion, kommer verktyget att anta att de två annotationerna är uppfyllda. När större programkod verifieras kan man även bevisa att de gäller även vid anrop till funktionen, genom att verktyget resonerar om den anropande funktionens beteende. Indatakontraktet innebär i detta fall att *n>0* och att minnet på adresserna *v ... (v+n-1)* är giltigt.

Nästa rad i kodkontraktet anger vad funktionen förväntas garantera vid avslut:

```
ensures \forall integer q; 0<=q<=n-1 ==> v[q]==(unsigned
char)0;
```

I vårt fall säger *ensures*-annotationen att minnet förväntas innehålla värdet *0* från och med adress *v* till och med adress *v+n-1*.

Vi har även en *assigns*-annotation som anger att minnet från och med adress *v* och *n* bytes framåt tilldelas av funktionen.

```
assigns v[0..n-1];
```

Vidare behöver WP hjälp med *while*-loopen och för den specificeras följande annotationer:

```
loop invariant 0 <= n <= \at(n, Pre);

loop invariant p == v+(\at(n, Pre)-n);
```

```
loop invariant \forall integer j;  0 <= j < (\at(n,
Pre)-n) ==> v[j] == (unsigned char)0;

loop assigns n, p, v[0..\at(n, Pre)-n-1];

loop variant n;
```

Dessa annotationer beskriver i princip det loopen gör, eftersom verktyget annars har svårt att räkna ut det själv. *\at(n,Pre)* är det värde *n* har när funktionen anropas. Den första invarianten anger att *n* kommer att ta värden mellan *0* och *\at(n,Pre)*. Den andra invarianten anger att, vid varje iteration, så är *p* lika med *v+(\at(n,Pre)-n)*. Observera att *n* minskar för varje iteration samtidigt som adressen som *p* pekar på ökar. Den tredje invarianten anger att det för varje iteration gäller att skrivna positioner i *v* är noll.

Loop assigns-annotationen anger vilka variabler som påverkas i loopen. I detta fall påverkas *n, p* och alla positioner i *v* som skrivits fram till och med denna iteration.

Loop-varianten är, som tidigare beskrivits, ett heltal som minskar vid varje loop-iteration och som är begränsad till positiva heltal. I detta fall är *n* ett sådant heltal.

För att köra verktyget användes följande kommando som innebär att verktyget öppnar sitt användargränssnitt och körs med insticksmodulen WP på filen aes.c:

frama-c-gui -wp -wp-rte aes.c

Filen *aes.c* innehåller funktionen *mbedtls_zeroise()* vilket är funktionen som ska testas. Argumentet *-wp-rte* innebär att RTE-modulen används för att automatiskt skapa annotationer för att fånga vanliga exekveringsfel.

5.3.6 Erfarenheter

Under arbetet med Frama-C har några observationer gjorts:

- Det kan behövas en hel del kodkontrakt och annotationer för att bevisa även relativt enkla funktioner, speciellt när det gäller funktioner som innehåller loopar och pekare.
- Kodkontrakten behöver korrekt beskriva funktionen för att Frama-C ska anse att funktionen uppfyller kodkontraktet. Det går inte att bevisa annoteringar i kodkontrakten en och en, vilket hade varit enklare då det hade underlättat felsökning. Små fel i en enda annotation gör att inget visas som uppfyllt.
- Det var inte enkelt att börja använda Frama-C, då det är komplicerat att förstå vad verktyget gör och hur det tolkar kodkontrakt och annoteringar. Det finns viss dokumentation och online-hjälp, men det krävs ändå

ganska mycket arbete att söka upp information. Det finns bildspel från presentationer med en del exempel, men det verkar saknas enkla kom-igång-exempel och handledningar.

- Den officiella kanalen för hjälp från skaparna av Frama-C är via internetformumet Stack Overflow[17]. Under arbetet undersöktes möjligheten att där få hjälp med ett problem som hade uppstått. Inom två dagar postades kod som löste det aktuella problemet.
- Verktyget utvecklas aktivt och har ett bra användargränssnitt. Det finns flera insticksmoduler som kan kombineras eller användas var för sig och det erbjuds på så sätt ett brett användningsområde.

5.4 CBMC

CBMC[18] (a Bounded Model Checker for ANSI-C and C++ programs) är som dess namn anger en bounded model checker för C/C++. CBMC är öppen källkod och relativt välspritt, bland annat finns verktyget som standard i Linux-distributionerna Debian, Ubuntu och Fedora. Verktyget introducerades 2004 och utvecklas kontinuerligt. Verktyget presterar alltjämt jämförbart med nyare och ofta mindre stabila forskningsprototyper. CBMC vann exempelvis 2014 års Software Verification Competition[19].

I detta avsnitt visar vi hur CBMC kan verifiera krav på funktionalitet och sekretess.

5.4.1 Verifiering av radering med CBMC

Vi vill verifiera att raderingsfunktionerna som har identifierats i avsnitt 4.2.1 är funktionellt korrekta. Vi illustrerar liksom i avsnitt 5.3.5 med *mbedtls_zeroize()*:

```
//Function under test
static void mbedtls_zeroize(void *v, size_t n) {
    volatile unsigned char *p = (unsigned char*)v;
    while (n--)
        *p++ = 0;
}
```

Figur 4. Funktionen *mbedtls_zeroize()* som ska verifieras.

[17] https://stackoverflow.com/tags/frama-c/ [läst 2018-07-05]
[18] https://www.cprover.org/cbmc/ [läst 2018-06-15]
[19] https://www.cs.ox.ac.uk/news/725-full.html [läst 2018-06-15]

Syftet med funktionen är att skriva över minnet med nollor från adress *v* till adress *v+n-1*. Vi kan fånga syftet med ett generellt test som anropar *mbedtls_zeroize()* med en godtycklig adress *v* och en godtyckligt längd *n* och sedan kontrollerar att det är nollor på relevanta adresser:

```
//Test of mbedtls_zeroize()
static void test_mbedtls_zeroize() {
  // Create arbitrary context for calling
  // function-under-test

  // Create an arbitrary buffer
  unsigned char v[SIZE];

  // Create an arbitrary cut-off point within buffer
  size_t n;
  // ... which is valid
  __CPROVER_assume(n < SIZE);
  __CPROVER_assume(n >= 0);

  // Call function-under-test
  mbedtls_zeroize(v, n);

  // Check that function-under-test has
  // intended zeroizing effect
  assert(isZero(v, n));
}
```

Figur 5. Generellt test av *mbedtls_zeroize()*.

Ber vi CBMC analysera testet *test_mbedtls_zeroize*() får vi beskedet i figur 6. Vi får veta att testet omvandlas till en satslogisk formel (”propositional reduction”) som beskriver alla möjliga indata som bryter mot assert-satsen sist i testet. Formeln skickas till en satslogisk teorembevisare (”MiniSAT”) som räknar ut att formeln inte är satisfierbar (”UNSATISFIABLE”), vilket betyder att assert-satsen är uppfylld oavsett minnesbuffertens initiala tillstånd (”VERIFICATION SUCCESSFUL”).

```
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
36132 variables, 94021 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.258s

** Results:
[test_mbedtls_zeroize.assertion.1] assertion isZero(v, n): SUCCESS

** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Figur 6. Verifikationsresultat.

Vi kan konstatera att funktionen som genomför testet, *test_mbedtls_zeroize()*, är mer omfattande än funktionen som analyseras, vilket förstås är avskräckande. Å andra sidan består den extra koden av rättfram programkod av ungefär samma slag som inom enhetstestning, som ju också har kritiserats just för att leda till alltför omfattande testkod.

En variant av säkerhetsegenskapen för nyckelradering kan vara att inte definiera ett specifikt överskrivningsvärde utan att istället definiera att överskriven data inte ska gå att återskapa. I ett sådant fall blir egenskapen som ska verifieras att de identifierade raderingsfunktionerna inte läcker känsliga data, varken explicit eller implicit via s.k. *covert channels*. Vi illustrerar med *mbedtls_cipher_free()*:

```
static void mbedtls_cipher_free(mbedtls_cipher_context_t *ctx)
{
    if (ctx == NULL)
        return;

    if (ctx->cipher_ctx)
        mbedtls_cipher_free(ctx->cipher_ctx);

    mbedtls_zeroize(ctx, sizeof(mbedtls_cipher_context_t));
}
```

Figur 7. Funktionen *mbedtls_cipher_free()* som ska verifieras.

Syftet med *mbedtls_cipher_free()* är återigen att aktivt radera minne, vilket i detta fall görs genom anropet till *mbedtls_zeroize()*.

Den variant av säkerhetsegenskapen som beskrivs ovan innebär dock ett mer abstrakt mål, nämligen att förstöra skyddsvärda data så att en angripare som råkar få tillgång till minnet inte kan återskapa dessa data, det vill säga tillståndet i

minnet (efter avslut) får inte avslöja den känsliga data som sparades i adressen *ctx*. En funktion klassas som sekretessbevarande om dess observerbara effekt inte beror på valet av hemlig indata (Goguen & Meseguer, 1984). Vi uttrycker denna variant av säkerhetsegenskapen med ett generellt test som kontrollerar att minnet efter avslut alltid innehåller samma sak, oberoende av indata, se figur 8. Det generella testet anropar *mbedtls_cipher_free()* med två godtyckliga känsliga indata, två *cipher_context:s*, och kontrollerar sedan att den observerbara effekten, minnesinnehållet efter avslut, är densamma i bägge fallen.

```
/*
Check that any two cipher_context:s leave indistinguishable memmory
after mbedtls_cipher_free()
*/
void test_weak_mbedtls_cipher_free()
{
    //Create two arbitrary contexts for calling function-under-test

        //Create an arbitrary cipher_context ctx1 in memory memory1
        mbedtls_cipher_context_t *ctx1 = construct_mbedtls_cipher_context(memory1);
        //Create an arbitrary cipher_context ctx2 in memory memory2
        mbedtls_cipher_context_t *ctx2 = construct_mbedtls_cipher_context(memory2);

    //Call function-under-test on both contexts

        //Call mbedtls_cipher_free on the first cipher_context
        mbedtls_cipher_free(ctx1);
        //Call mbedtls_cipher_free on the second cipher_context
        mbedtls_cipher_free(ctx2);

    //Check that the observable effect is the same

        //Check that memory1 and memory2 are industinguishable
        assert(equals(memory1, memory2));
}
```

Figur 8. Generellt test av *mbedtls_cipher_free()*.

Ber vi CBMC analysera testet *test_mbedtls_cipher_free()* får vi beskedet att verifikationen misslyckas! Ber vi så verktyget om motexempel finner vi att längden på det hemliga *cipher_context* syns i minnet efter avslut.

Låt oss anta att det identifierade informationsläckaget inte spelar någon roll, dvs. att vår säkerhetsegenskap var för stark. Vi vill tillåta att en viss specifik aspekt hos hemlig data röjs, nämligen längden, men inget annat. Vi uttrycker denna svagare säkerhetsegenskap med ett generellt test som kontrollerar att all känslig indata av samma längd har samma observerbara effekt, se figur 9. Det generella testet anropar nu *mbedtls_cipher_free()* med två godtyckliga känsliga indata *av samma längd* och kontrollerar sedan att den observerbara effekten, minnet efter avslut, är densamma i bägge fallen.

```
/*
Check that cipher_context:s of equal length (depth) leave indistinguishable memmory
after mbedtls_cipher_free(ctx)
*/
void test_weak_mbedtls_cipher_free()
{
    //Create two arbitrary contexts for calling function-under-test

        //Create an arbitrary cipher_context ctx1 in memory memory1
        mbedtls_cipher_context_t *ctx1 = construct_mbedtls_cipher_context(memory1);
        //Create an arbitrary cipher_context ctx2 in memory memory2
        mbedtls_cipher_context_t *ctx2 = construct_mbedtls_cipher_context(memory2);

    //Consider only cipher_contexts of equal length (nesting depth)

        //Abort if cipher contexts are of different length
        __CPROVER_assume(len(ctx1) == len(ctx2);


    //Call function-under-test on both contexts

        //Call mbedtls_cipher_free on the first cipher_context
        mbedtls_cipher_free(ctx1);
        //Call mbedtls_cipher_free on the second cipher_context
        mbedtls_cipher_free(ctx2);

    //Check that the observable effect is the same

        //Check that memory1 and memory2 are industinguishable
        assert(equals(memory1, memory2));
}
```

Figur 9. Svagare generellt test av *mbedtls_cipher_free()*.

Vi ber åter CBMC att analysera testkoden, *test_weak_mbedtls_cipher_free()*, varvid verifieringen denna gång lyckas.

5.4.2 Erfarenheter

Verifiering med model checking beskrivs ofta som bara en knapptryckning bort. I vårt fall krävde dock CBMC mer än bara en knapptryckning. Framförallt gick det åt tid till att bryta ned krav till tester på enskilda funktioner. Vi kan också konstatera att testkoden ovan är mer omfattande än funktionen som analyserades.

I en annan mening var CBMC dock så enkelt som utlovat. All matematisk logik är automatiserad och gömd för användaren. Funktionella krav fångas med test-kod i C och feedback från verktyget utgörs av konkreta testdata, exempelvis i form av specifik indata som påvisar fel i den testade koden.

5.5 KLEE

KLEE[20] är ett välspritt verktyg som har spelat en viktig roll i det senaste decenniets renässans för symbolisk exekvering. Verktyget har vunnit flera akademiska utmärkelser (bl.a. ACM Computer and Communications Security Test of Time Award) och refereras till flitigt (över 1700 referenser). Verktyget är öppen källkod och används i viss utsträckning som ramverk för att implementera olika former av symbolisk exekvering.

5.5.1 Användning av KLEE

Verktyg för model checking används alla på liknande sätt. Användningen av KLEE och CBMC skiljer sig i stort sett åt endast i fråga om hur symboliska indata definieras, det vill säga hur användaren anger rymden av möjlig indata som ska genomsökas. I CBMC är initiala värden på oinitierade variabler symboliska och därmed obestämda. I KLEE behöver användaren explicit deklarera en variabel som symbolisk med en särskild programinstruktion. Vi illustrerar med *test_mbedtls_zeroize()* (från avsnitt 5.4.1) som testar att *mbedtls_zeroize(void *v, size_t n)* verkligen skriver över minnet med nollor från och med adress *v* och *n* bytes fram, se figur 10. Vi har lagt till två *klee_make_symbolic*-instruktioner, i övrigt är testkoden densamma som för CBMC.

Ber vi KLEE analysera testkoden för alla möjliga tillstånd i minnesbufferten får vi återigen beskedet att assert-satserna alltid är uppfyllda oavsett minnesbuffertens initiala tillstånd.

5.5.2 Erfarenheter

Vi kan notera att beräkningen gick avsevärt mycket snabbare än med CBMC, vilket är i linje med teorin för model checking. I övrigt skilde sig inte användningen av KLEE och CBMC åt nämnvärt. Trots att verktygen använder helt olika slags model checking kunde analysen från avsnitt 5.4.1 genomföras nästan på samma sätt med KLEE som med CBMC.

```
static void test_mbedtls_zeroize() {
  // Create arbitrary context for calling
  // function-under-test

  // Create a buffer
  unsigned char v[SIZE];
```

[20] https://klee.github.io/ [läst 2018-06-15]

```
    // ... and make it arbitrary
    klee_make_symbolic(v, SIZE, "v");

    //Create a cut-off point within buffer
    size_t n;
    // ... and make it arbitrary
    klee_make_symbolic(&n, sizeof(n), "n");
    // ... but valid
    klee_assume(n < SIZE);
    klee_assume(n >= 0);

    //Call function-under-test
    mbedtls_zeroize(v, n);

    // Check that function-under-test has intended
    // zeroizing effect
    assert(isZero(v, n));
}
```

Figur 10. Generellt test av *mbedtls_zeroize()* anpassat för KLEE.

5.6 CodeSonar

CodeSonar[21] från GrammaTech är ett kommersiellt verktyg för statisk analys av både källkod och binärkod. CodeSonar stödjer källkodsanalys i språken C, C++ och Java.

Verktyget stödjer analys enligt ett antal fördefinierade regler med möjlighet att skriva egna analysregler. Reglerna som finns implementerade i CodeSonar kommer bland annat från följande uppsättningar:

- Motor Industry Software Reliability Association (MISRA) C[22]
- SEI CERT[23] C Coding Standard, C++ Coding Standard
- Defence Information Systems Agency (DISA) Security Technical Implementation Guides (STIG)[24]
- Common Weakness Enumeration (CWE)[25].

[21] https://www.grammatech.com/products/codesonar [läst 2017-12-11]
[22] https://www.misra.org.uk/ [läst 2017-12-11]
[23] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards [läst 2017-12-11]
[24] https://iase.disa.mil/stigs/Pages/index.aspx [läst 2017-12-11]
[25] https://cwe.mitre.org/ [läst 2017 12 11]

Information om vilka tekniker och metoder som verktyget använder sig av är inte publikt tillgänglig, annat än på en mycket övergripande nivå. På GrammaTechs webbsidor[26] om CodeSonar beskrivs att verktyget bland annat använder symbolisk exekvering för att utforska vägar genom programmet och relationer mellan variabler i programmet. CodeSonar stödjer även kontamineringsanalys där variabler som kan innehålla data från externa källor markeras och kan följas genom källkoden.

5.6.1 Användning av CodeSonar

CodeSonars centralpunkt är en hub där analysresultat samlas i en databas och presenteras via ett webbgränssnitt. Analyserna initieras av exempelvis utvecklare eller byggserver och startas genom att mjukvaran byggs via CodeSonars analysverktyg. Analysverktyget startas med kommandot *codesonar analyze*:

```
codesonar analyze <namn> <hubadress> <byggkommando>
```

För analys av mbed TLS kan kommandot exempelvis bli:

```
codesonar analyze mbedtls cshub.foi.se:7340 make
```

Detta kommando använder det ordinarie byggsystemet, genom kommandot *make*, för att genomföra en likvärdig byggprocess där CodeSonar fångar upp den kod som byggs och skapar en modell av denna i en intern representation. Den interna representationen används sedan i analysen, där ett antal regler undersöks enligt förvalda filter. Analysen kan distribueras över ett *analysmoln*, där många datorer i samma nätverk kan samverka för en snabbare analys av koden.

I webbgränssnittet går det exempelvis att se aktuella varningar, hur varningarna har förändrats mellan analystillfällen, komplexitetsmått samt visualiseringar av koden. För varningarna ges en annoterad kodvy med expanderade funktionsanrop, där varningen visas tillsammans med indata och beslutsvägar (villkorssatser) som leder till det konkreta fall som varningen gäller.

5.6.2 Regler

Säkerhetsmål 2 (se avsnitt 4.1.2) har varit fokus vid användning av CodeSonar. För att matcha de valda säkerhetsegenskaperna, i form av regler från SEI CERT C Coding Standard, har passande filter valts ut i CodeSonar. Spårningen från regeluppsättningarna till filter återfinns i överskådlig form i dokumentationen som följer med CodeSonar.

[26] https://www.grammatech.com/products/source-code-analysis [läst 2017-12-11]

Tabell 2 visar de utvalda filtren i CodeSonar samt hur många varningar som ges för respektive filter när de körs på den kombinerade källkoden för OpenVPN och mbed TLS, totalt omfattande drygt 100 000 rader källkod.

Tabell 2. Regler, filter och antal varningar vid analys av OpenVPN och mbed TLS. Beskrivning av de olika reglerna finns i avsnitt 4.2.2.

Regel	Filter	Antal varningar
ARR30-C	Buffer overrun	35
EXP33-C	Unitialized variable	6
MEM30-C	Use after free	2
MEM31-C	Leak	7
STR31-C	No space for null terminator	5
Totalt		55

CodeSonar kategoriserar varningarna samt rankar dem utifrån en poäng som sätts i förhållande till hur allvarlig varningen är enligt verktyget. Leak-varningarna tillhör kategorin *tillförlitlighet* (eng. reliability) och rankas i detta fall högst. Övriga varningar återfinns i kategorin *säkerhet* (eng. security).

Efter att varningarna har presenterats i webbgränssnittet återstår ett manuellt arbete med att gå igenom dessa för att se vilka som är sant positiva och kan behöva åtgärdas. Det manuella arbetet utgår från de specifika varningarna och den aktuella koden, vilket kräver en kompetent programmerare.

Av de 55 varningarna har den manuella genomgången endast visat på sju garanterat falskt positiva resultat. Övriga varningar har, med varierande sannolikhet, ansetts vara sanna positiva resultat. Åtta av varningarna pekar på potentiella buggar som kan påverka funktionaliteten.

5.6.3 Erfarenheter av CodeSonar

Överlag har CodeSonar varit smidigt att använda och verktyget ger intryck av att vara moget, med bra dokumentation och smidig hantering såväl vid installation som vid användning. För testobjektet har integrationen med byggsystemet varit smidig och gjort analysprocessen i princip helt smärtfri.

Det enda problem som vi stötte på i analysen var att CodeSonar inte stödde den nya flyttalstypen _Float128 som används i vissa biblioteksfunktioner, dock gick detta problem relativt lätt att kringgå. När detta skrivs har en ny version av

CodeSonar släppts som troligtvis har stöd för datatypen, vilket i så fall löser problemet.

Varningarna presenteras på ett överskådligt sätt som är lätt att följa genom källkoden. Då varningarna i flera fall gäller komplexa beroenden eller skeenden i mjukvaran upplever vi att det krävs en erfaren och kompetent programmerare för att förstå vissa av varningarnas bakomliggande orsaker till fullo.

En nackdel med CodeSonar i denna rapports kontext är bristen på information om de metoder som verktyget använder internt. Det vore önskvärt att få mer information om vilka principer som ligger bakom analyserna för att kunna utvärdera verktygets styrka och i vilken utsträckning som analyserna hittar de kategorier av programmeringsfel som söks.

6 Diskussion

Korrekt mjukvara avseende både funktionalitet och säkerhet är avgörande eftersom den kan påverka exempelvis personsäkerhet eller skyddsvärd information. Verifiering av mjukvara är ett komplext problem, där många olika metoder och verktyg måste samverka för att nå en hög tilltro till att mjukvaran är korrekt. Verifiering innebär att såväl specifikation som implementation undersöks på djupet.

Denna studie har undersökt användbarhet och funktion hos verktyg som använder formella metoder för verifiering av mjukvara skriven i programspråket C. Inom mjukvaruområdet används termen *formella metoder* för att beteckna en bred kategori av matematiska tekniker som i sina olika former kan användas för att producera olika typer av utsagor gällande källkodens funktion eller överensstämmelse med en specifikation.

Arbetet i studien har centrerats kring att verifiera några utvalda säkerhetsegenskaper, det vill säga enskilda aspekter av IT-säkerhetsfunktioner, baserat på två säkerhetsmål på hög nivå. Det har inte funnits någon intention att genomföra en komplett verifiering, utan dessa säkerhetsegenskaper har använts för att ha konkreta och relevanta exempel att undersöka med de verktyg som har utvärderats i studien.

När de utsagor som verktygen producerar handlar om överensstämmelse med en specifikation utgör utsagorna ett slags bevis på att koden uppfyller specifikationen. Specifikationen beskriver en viss programfunktion givet ett visst (annat) villkor på indata. Då bevisen endast är giltiga inom de antaganden och förutsättningar som ges av specifikation och verktyg, är det nödvändigt att förstå dessa avgränsningar för att kunna förstå bevisens styrka. I vissa fall kan det vara svårt att genomskåda alla antaganden och förutsättningar då de är implicita, exempelvis i form av val av verktyg eller inställningar i verktyget.

Formella metoder har stor spännvidd i fråga om bevisförmåga, vilket naturligt avspeglas i styrkan hos de resultat som produceras av verktygen. Bland de verktyg som har utvärderats i denna studie syns även en koppling mellan utsagornas styrka och hur komplicerat det är att använda verktygen. I ena änden av skalan återfinns CodeSonar, ett verktyg som kombinerar formella och icke-formella analysmetoder. Verktyget ger dock inte några garantier för varningarnas riktighet. I studien har verktyget upplevts vara lätt och snabbt att använda även på större kodmängder. I andra änden finns Frama-C med modulen ”minst begränsande förvillkor”, som ger garantier för att givna kodkontrakt uppfylls. Denna modul kräver dock god kunskap och en relativt omfattande arbetsinsats för varje funktion och aspekt som ska bevisas.

En parameter som påverkar valet av verktyg och som har visat sig vara viktig under studien är möjligheten att kunna få support. Vid användningen av

Frama-C:s modul för minst begränsande förvillkor visade sig tillgången till support vara avgörande för att kunna skriva korrekta kodkontrakt och annotationer för att få fram ett bevis. Sannolikt kommer behovet av support att minska med större erfarenhet, men risken är ändå stor att vissa kodkonstruktioner kräver komplexa kodkontrakt som är svåra att skriva. I fallet med Frama-C understryks behovet av support ytterligare av att alla kontrakt och annotationer måste vara uppfyllda för att verktyget ska producera ett bevis. Värdet av en supportfunktion eller ett aktivt användarforum är därmed stort.

Bland de formella metoder som inte har behandlats i denna studie finns ett antal olika tekniker som verkar kunna uppfylla några av studiens specifika verifieringsmål. Ett exempel är kontamineringsanalys som kan ge information om hur data som är skyddsvärd eller skadlig propagerar genom mjukvaran. En väl genomförd och komplett kontamineringsanalys ger en större tilltro till att alla relevanta delar av källkoden, dvs. där potentiellt skyddsvärd eller skadlig data hanteras, har identifierats.

Formella metoder är i dagsläget inte någon praktisk totallösning för att ersätta traditionell programvarutestning och kommer sannolikt inte heller att vara det inom överskådlig tid. Detta eftersom det idag inte är praktiskt möjligt att formellt bevisa funktionaliteten hos ett helt program förutom i mycket enkla fall. Istället är metoderna ett komplement till traditionella verifieringstekniker, som exempelvis kodgranskning, enhetstester, integrationstester, fuzz-tester och penetrationstester. Tillsammans kan teknikerna ge en hög grad av täckning i verifieringsarbetet och därmed ge hög tilltro till mjukvaran.

Med formell verifiering överlåts en hel del av förtroendeskapandet till verktygen som tar fram bevisen för att koden överensstämmer med specifikationen. För att kunna ha hög tilltro till bevisen krävs därmed validering av såväl specifikationer som verktyg. Specifikationerna (exempelvis i form av kodkontrakt) måste överensstämma med den tänkta funktionen i mjukvaran och verktygen måste utföra bevisföringen korrekt. Användningen av formella metoder flyttar därmed en del av arbetet från verifiering av källkoden till exempelvis validering av verktygen. Med en genomtänkt strategi för införande av verktygsstöd i utvecklings- och granskningsarbetet kan kostnaden fördelas över de mjukvaruprojekt där verktygen används.

Verktygen och deras inställningar har stor inverkan på analysresultatet, exempelvis genom att analyser tas med eller inte, som i fallet med analysfilter i CodeSonar, eller genom att bevisförmågan påverkas, som i valet av minnesmodell i Frama-C. Det är således mycket viktigt att förstå verktygen och de begränsningar de för med sig för att kunna bedöma resultatens värde i det sammanlagda verifieringsarbetet.

En insikt under studien har varit att alla verktyg för formella metoder inte nödvändigtvis lämpar sig för verifiering av befintlig kod. Exempelvis saknas

det ofta tillräckligt detaljerad dokumentation om det testade systemet för att kunna skapa tillförlitliga kodkontrakt. De verktyg som ligger närmare källkoden, exempelvis med behov av annotationer och kodkontrakt, för med sig ett större behov av att källkoden är skriven på lämpligt sätt. Till exempel finns kodkonstruktioner som kan vara svåra att bevisa med verktyg som använder formella metoder, trots att en utvecklare relativt enkelt kan förstå koden och skriva om den till en form som är lätt att bevisa. Om verktygsstödet tas in under programmeringsfasen kan källkoden utformas och anpassas så att verktygen blir lätta att använda.

Ett ytterligare problem när befintlig källkod verifieras är att det krävs god förståelse för kodens uppbyggnad för att kunna genomföra en effektiv och vältäckande verifiering. Även om källkoden som har använts i studien överlag gav intryck av att vara välskriven så var den inte designad med verifierbarhet som huvudmål. Dessutom fanns det endast begränsad dokumentation, vilket gjorde att koden delvis var svårgenomtränglig. Nedbrytning i säkerhetsegenskaper och spårning av vad som faktiskt verifieras skulle underlättas avsevärt om arkitektur och källkod togs fram med spårbarhet och verifieringsarbete i åtanke.

7 Resultat och slutsatser

Det är viktigt att förstå de begränsningar som finns med formella metoder. En del i mognandet vad gäller att använda formella metoder under utveckling är att förstå dessa begränsningar. Med kunskap och erfarenhet kanske den viktigaste insikten handlar om vad som är lämpligt att verifiera med formella metoder och vad som inte är möjligt eller lämpligt att verifiera.

Användning av formella metoder är förtroendehöjande – tilltron till mjukvarans korrekthet ökar – men ger i allmänhet inte absoluta bevis. När de är som bäst kan formella metoder ge ett väldigt starkt förtroende för det som bevisas, men det är sällan genomförbart att producera ett absolut bevis. Bevisen bygger alltid på någon form av antaganden som begränsar bevisens värde, där dessa antaganden kan återfinnas i såväl specifikationer som verktyg.

Det finns därför ett behov av att validera såväl kedjan från krav till specifikationer som de verktyg som används. Specifikationer i form av kodkontrakt som ställer krav på funktionen kan baseras på felaktiga antaganden eller felaktig tolkning av den högre nivå av kravställning på programmet som finns. Även verktygen som används för att tillämpa formella metoder behöver valideras. Detta för att säkerställa att verktygen har korrekt funktion och därmed ökar tilltron till systemet.

Verifiering av att en funktion motsvarar det som har kravställts måste alltid utföras för att säkerställa efterfrågad funktionalitet. Formella metoder som kräver specifikationer i form av kodkontrakt underlättar verifieringen av implementationen av funktionen jämfört med att granska koden eftersom kodkontraktet erbjuder en tydligare beskrivning som gör det enklare att hitta felaktiga antaganden. Genom att använda verktyg för formella metoder kan funktionen sedan automatiskt verifieras mot specifikationen.

Som alltid är alltså kunskap, strukturerat arbete och planering viktiga inslag i verifieringsarbetet, även när formella metoder används för att ta fram bevis för en mjukvaras korrekthet eller för att hitta fel i koden. Med lämplig metodik och kunnig personal menar vi att användning av verktyg för formella metoder ger ett mervärde vid verifiering av säkerhetskritisk mjukvara, speciellt när metoderna används redan under programmeringsfasen.

7.1 Rekommendationer

Studien har visat att formella metoder i många fall är praktiskt användbara, både avseende bevisförmåga och arbetsinsats. Lättviktiga verktyg, som exempelvis CodeSonar, är så lättanvända att de är lämpliga att direkt integrera i det vanliga arbetsflödet för utvecklare och bör därmed alltid användas. Den typen av verktyg

ger förvisso inga garantier för kodens kvalitet, men de gör att utvecklarna kan flytta sitt fokus från undvikande av enkla fel till svårare problem.

Mer tungviktiga verktyg för formella metoder är svåra att använda vid granskning, om koden inte har skrivits specifikt för att kunna verifieras med sådana verktyg. En bättre lösning är att använda verktygen redan under programmeringsfasen i utvecklingen, för att på så sätt direkt kunna skriva koden på lämpligt sätt. Dessutom ger det möjlighet att iterativt anpassa koden efter verktygen så att maximal bevisbarhet kan nås på ett effektivt sätt.

Ett sätt att introducera formella metoder i utvecklingsprocessen vore att initialt endast applicera dem på avgränsade delar av systemet för att få en måttlig kodmängd att beakta under inlärningsfasen. När erfarenhet av att använda metoderna och verktygen har byggts upp går det att utöka omfattningen till en större del av källkoden.

7.2 Sammanfattning av resultat

I tabellerna nedan kopplas studiens resultat till frågorna i avsnitt 1.1.

Finns det verktyg som kan användas för att verifiera (delar av) säkerhetsmålen i studien?	
Säkerhetsmål 1, radering av nyckelminnet	CBMC, KLEE, Frama-C
Säkerhetsmål 2, kodkvalitet	CodeSonar
Kommentar: Svaren i tabellen ovan visar endast denna studies urval och beskriver inte verktygens hela potential. Det är mycket viktigt att förstå verktygen och de begränsningar som de för med sig för att kunna bedöma verktygens resultats värde i det sammanlagda verifieringsarbetet. Formella metoder bör inte appliceras på befintlig kod, utan bör istället redan när systemet utvecklas. Undantaget är CodeSonar, som kan användas som en del av granskningsarbetet.	

Vilken bevistyp ger verktygen? Dvs. ger verktyget bevis för korrekthet eller bevis för förekomst av en typ av fel?	
CBMC	Producerar bevis för funktioners korrekthet mot specifikation. Kan bevisa förekomst eller avsaknad av vissa typer av fel.
KLEE	Producerar bevis för funktioners korrekthet mot specifikation. Kan bevisa förekomst eller avsaknad av vissa typer av fel.
Frama-C	Producerar bevis för funktioners korrekthet mot specifikation. Kan bevisa förekomst eller avsaknad av vissa typer av fel.
CodeSonar	Pekar ut förekomst av vissa typer av fel. Kan ej påvisa avsaknad av fel. Det saknas information om de formella metoder som verktyget använder internt.

Vilken är mognadsgraden hos verktygen med avseende på installation och support?	
CBMC	• Komplicerad installationsprocess • Har ej testat support
KLEE	• Komplicerad installationsprocess. • Har ej testat support.
Frama-C	• Komplicerad installationsprocess. • God användarhandledning. • Bra supportforum.
CodeSonar	• Välfungerande installationsprocess • God användarhandledning • Bra support

Hur komplext är det att använda verktygen? Vilken kompetens krävs för att använda verktygen?	
CBMC	Alla tre verktygen kräver kunskap och förståelse för formella metoder för att kunna användas. Det krävs även en relativt omfattande arbetsinsats för varje funktion och aspekt som ska bevisas.
KLEE	
Frama-C	
CodeSonar	Resultat med en knapptryckning. Kräver ingen djupare kunskap om formella metoder. Dock krävs kvalificerad kodgranskning för bedömning av varningar.

Referenser

Baudin, P., Bobot, F., Correnson, L., & Dargaye, Z. (2016). *WP Plug-in Manual.* Retrieved from https://frama-c.com/: https://frama-c.com/download/wp-manual-Silicon-20161101.pdf

Bühler, D., Cuoq, P., & Yakobowski, B. (2016). *EVA - The Evolved Value Analysis plug-in.* Retrieved from https://frama-c.com: https://frama-c.com/download/value-analysis-Silicon-20161101.pdf

Cousot, P., & Cousot, R. (1997). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.*

Godefroid, P. (2015). Between Testing and Verification: Software Model Checking via Systematic testing. *Haifa Verification Conference.*

Goguen, J., & Meseguer, J. (1984). Unwinding and Inference Control. *Security and Privacy.*

Harman, M., & Hierons, R. (2001). An overview of program slicing. *Software focus*, 85–92.

Herrmann, P., & Signoles, J. (2017). *Frama-C's annotation generator plug-in.* Retrieved from https://frama-c.com: https://frama-c.com/download/rte-manual-Sulfur-20171101.pdf

IntelliTest. (2016). *Generate unit tests for your code with IntelliTest.* Retrieved from microsoft.com: https://msdn.microsoft.com/en-us/library/dn823749.aspx

King, J. (1976). Symbolic Execution and Program testing. *Communications of the ACM.*

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2015). Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 573–609.

Lomuscio, A., Qu, H., & Raimondi, F. (2017). MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*.

Meyer, B. (1997). *Object-Oriented Software Construction.* Prentice Hall.

MISRA. (2012). *Guidelines for the Use of the C Language in Critical Systems.* The Motor Industry Software Reliability Association.

Sabelfeld, A., & Myers, A. (2003). A Model for Delimited Information Release. *Software Security - Theories and System.*

SEI CERT. (2016). *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems.* Carnegie Mellon University.

Tillmann, N., & Schulte, W. (2005). Parameterized unit tests. *10th European software engineering conference.* ACM SIGSOFT Software Engineering Notes.

TrustInSoft. (u.å.). *PolarSSL 1.1.8 verification kit.* Retrieved from https://trust-in-soft.com: https://trust-in-soft.com/polarSSL_demo.pdf

Weiser, M. (1981). Program Slicing. *5th International Conference on Software Engineering* (pp. 439–449). San Diego, California, USA: IEEE Press.

Ye, K. Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., & Appel, A. W. (2017). Verified Correctness and Security of mbedTLS HMAC-DRBG. *ACM SIGSAC Conference on Computer and Communications Security* (pp. 2007–2020). Dallas, Texas, USA: ACM.

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