FOI

Self-healing and self-protecting software

Scanning the research frontier

IOANA RODHE, ERIK WESTRING, HENRIK KARLZÉN

FOI-R--3836--SE
ISSN 1650-1942 January 2014

Ioana Rodhe, Erik Westring, Henrik Karlzén

Self-healing and self-protecting software

Scanning the research frontier

Bild/Cover: Aureliy Movila

Titel	Självläkande mjukvara – skanning av forskningsfronten
Title	Self-healing and self-protecting software – scanning the research frontier
Rapportnr/Report no	FOI-R--3836--SE
Månad/Month	Jan/Jan
Utgivningsår/Year	2014
Antal sidor/Pages	39
ISSN	1650-1942
Kund/Customer	Internt
Forskningsområde	4. Informationssäkerhet och kommunikation
FoT-område	Avskanning av forskningsfronten
Projektnr/Project no	I35412
Godkänd av/Approved by	Christian Jönsson
Ansvarig avdelning	Informations- och aerosystem

Sammanfattning

Det blir allt svårare att skydda växande och komplexa IT-system enbart med hjälp av proaktiva säkerhetsmekanismer som hanterar tidigare kända hot och fel. Konceptet självläkande mjukvara (eng. self-healing and self-protecting software) har därför framkommit som ett svar på behovet av drift- och säkerhetslösningar som kan tillämpas vid körning av ett system för att identifiera nya eller okända hot och fel, samt försöka fixa dessa faror dynamiskt. Vidare låter självläkande mjukvara tillgängligheten gå före korrigering av mindre allvarliga problem. Detta är av särskilt intresse för verksamhetskritiska system och kan tillämpas i system som kräver hög nivå av autonomitet, såsom rymdsonder och obemannade farkoster.

I denna rapport studerar vi konceptet självläkande mjukvara som en del av visionen om det autonoma datorsystemet. Vi ger en introduktion och en bakgrund till konceptet samt fastställer lämplig definition och diskuterar närliggande koncept i det allmänna forskningsområdet. Vi presenterar också intressant forskning kring och tillvägagångsätt för självläkande mjukvara, med lämpliga kategoriseringar.

Nyckelord: självläkande, litteraturstudie, autonoma datorsystem, skannande, forskningsfronten

Summary

It is getting more and more difficult to protect increasingly complex IT systems by only using proactive security mechanisms that deal with known threats and faults. The concepts of self-healing and self-protecting software (SHASP) have emerged from the need for security and safety solutions that can be applied to running systems in order to identify new and unknown threats and failures and to try to fix these threats dynamically. Furthermore, SHASP lets system availability take precedence over fixing less important problems. This is of particular interest for mission-critical systems and may find applications in systems that require high autonomy, such as space probes and unmanned vehicles.

In this report we study the concepts of self-healing and self-protection as a part of the autonomic computing vision. We give an introduction and a background for the two concepts and establish appropriate definitions and discuss adjacent concepts in the general research area. We also present interesting research and approaches to self-healing and self-protection with suitable categorisations.

Keywords: Self-healing, self-protection, literature review, autonomic computing, SHASP, scanning, frontier

Table of contents

1 Övergripande sammanfattning

Detta kapitel utgör en sammanfattning av rapporten på svenska. Rapporten i övrigt är skriven på engelska.

Syfte och mål

Syftet med att skanna forskningsfronten är att hitta nya intressanta forskningsområden som har potential att vara till nytta för FOI och dess kunder. Avskanningen utgör därmed en del av FOI:s strategiska förnyelse. Ett möjligt nytt forskningsområde är självläkande mjukvara (SM; eng. self-healing and self-protecting software, SHASP). SM är autonoma mjukvarudelar som kan detektera och hantera nya, tidigare okända, problem vid körning. Problemen kan vara i form av angrepp från illasinnade eller buggar i programkoden, som ger upphov till felaktigheter. SM kan därför förbättra dagens datorsystem på flera sätt. Mjukvara i allmänhet kan bli bättre testad och därmed innehålla färre buggar, vara lättare att hantera för användare och administratörer, samt bli mer motståndskraftig mot angrepp. Särskilt intressant är självläkande mjukvara för system med höga krav på tillgänglighet eller autonomt beteende, såsom rymd- och försvarsapplikationer eller kritisk distribuerad infrastruktur.

I denna rapport presenterar vi en litteraturstudie som identifierar relevant forskning inom området. Målet är att utforska och ge insikt i ett forskningsområde som tidigare kanske inte fått så mycket fokus som det förtjänar. Denna rapport kan därför utgöra en lämplig startpunkt för mer detaljerat arbete. Bakgrunden till området, tillsammans med vanliga definitioner, avgränsning samt relevanta bidrag presenteras i denna rapport.

Bakgrund

Idén om autonoma datorsystem (eng. autonomic computing) uppstod som ett svar på den ökande komplexiteten hos systemen och behovet av att ta en helhetsgrep på hanteringen av dessa. Genom den ökade användningen av webben, växte antalet felkällor och därmed också behovet av snabbt beslutsfattande kraftigt. Administratörerna var överansträngda, system och mjukvara innehöll för mycket kod för att som helhet kunna överblickas av människor och datoriseringen i samhället ökade. Allt detta gav upphov till IBM:s idé om datorer som kunde ta hand om sig själva i högre utsträckning (Kephart 2003). Ett autonomt datorsystem måste känna sig självt, sina begränsningar och sin omgivning. Det bör dessutom dölja komplexitet från, och anpassa sig till, användaren. Allt detta innebär att system måste konfigurera sig självt, optimera sig självt samt läka och skydda sig självt. De två sista egenskaperna, vilka denna rapport fokuserar på, benämns fortsättningsvis gemensamt som självläkande

mjukvara, eftersom de har mycket gemensamt. En tidig (fritt översatt) definition gavs av Kephart (2003):

”När självläkande mjukvara angrips eller drabbas av såväl nya som gamla fel, kan den autonomt och dynamiskt (genom realtidsbeslut) upptäcka samt diagnostisera problemet och läka de skador som uppstått.”

Den ursprungliga inspirationen till självläkande mjukvara och autonoma datorsystem kom från biologin, mer specifikt människans autonoma nervsystem och dess nervceller. Det är dock inte helt uppenbart hur liknelsen mellan datorsystem och nervsystem var tänkt och även om det inte var IBM:s intention att skapa fullständig artificiell intelligens, så bör det noteras att det är en obesvarad forskningsfråga om detta alls är möjligt.

En angränsande liknelse, med människans immunförsvar som grund, ligger kanske närmare till hands. Till exempel är begreppet datorvirus inspirerat av den biologiska motsvarigheten och även om liknelsen inte är perfekt så är den intressant och har uppenbarligen varit framgångsrik när det gäller antivirusskydd.

Det mänskliga immunförsvaret kan delas in i två delar. Den första hanterar hot på en ganska rudimentär nivå genom exempelvis den fysiska gränsen som huden utgör, kemiskt skydd genom saliven och magsyran, eller genom feber och andra inflammatoriska mekanismer. Den andra delen av immunförsvaret är mer sofistikerad och består av specifika samt adaptiva försvarsmekanismer. En grupp av antikroppar diagnostiserar främmande livsformer, medan en annan grupp reagerar och läker (The History of Vaccines 2013). Denna ansvarsfördelning ger visst skydd mot så kallad autoimmunitet där kroppen angriper sig själv. Reaktioner hindras såvida inte diagnostiska antikroppar varit inblandade, samt utifrån av immunförsvaret införskaffad diagnostisk kunskap från vaccinering, blodtransfusioner, arv och tidigare angrepp. Ytterligare en spärr kan utgöras av att kroppen kräver signaler på att något faktiskt skadats innan den låter antikroppar rycka in (se till exempel Swimmer 2007).

Definition

Självläkande tekniker fokuserar på att hålla mjukvaran igång, även vid mindre allvarliga fel och angrepp, istället för att tvingas stoppa och laga eller rent av krascha. Dessutom begränsas spridningen av eventuella angrepp. Fokus ligger på att hantera okända (nya) fel och angrepp. Eftersom det finns andra säkerhetsmekanismer, såsom intrångsdetektering, som detekterar angrepp, har denna rapport inget större fokus på just den delen av självläkande mjukvara.

SM har de fyra livscykelstadierna *övervaka*, *analysera* (ett upptäckt problem), *planera* och *verkställa* (en avhjälpande åtgärd). En distinkt egenskap hos SM är att de främst hanterar nya och okända säkerhetsproblem och fel. Då problemen är nya, och framförallt eftersom mjukvaran måste undersöka sig själv, undviks

proaktiva lösningar till förmån för reaktiva. Vidare hanteras problemen av ett delvis eller helt autonomt och dynamiskt system där beslut om läkande tas med minimal mänsklig inblandning och utan att den inblandade komponenten först behöver stängas av. Detta är särskilt användbart i realtidssystem där tillgänglighet har högst prioritet. Vidare reparerar SM komponenter istället för att byta ut dem mot säkerhetskopior eller reservdelar. Detta är särskilt viktigt för att kunna hantera tidigare okända problem. Slutligen integreras vanligen försvarsteknikerna i existerande system, snarare än att implementeras redan i designfasen, men detta innebär också vissa begränsningar gällande precisionen hos mekanismerna.

SM mekanismerna avses inte utgöra något slutgiltigt svar på alla säkerhetsproblem och andra fel i mjukvara, eller att ens ersätta befintliga lösningar, utan att komplettera och att finnas som en extra resurs i det fall allt annat fallerar. Som nämnts ovan elimineras inte mänsklig inblandning helt, utan reduceras enbart. Dessutom behöver många av de i litteraturen föreslagna teknikerna en inlärningsperiod för att uppnå självkännedom, varför sådana tekniker presterar bättre och bättre med tiden. Många hot har nya egenskaper men också mycket gemensamt med tidigare, kända, hot. Genom att placera hoten i olika klasser kan självläkande mjukvara reagera mer effektivt. Till exempel kan två skilda fel ge samma konsekvens, såsom att nätverket kraschar eller att en angripare kan överskrida en minnesbuffert och därmed ta kontroll över systemet.

Avgränsning

Det finns flera mjukvarurelaterade försvarstekniker som liknar självläkande mjukvara men har ett litet annat fokus, även om de ibland överlappar och har gränser som inte alltid är uppenbara. Här följer två exempel.

Antivirusprogram *"övervakar en dator eller ett nätverk för att identifiera alla viktigare typer av skadlig kod samt för att förhindra incidenter och spridning av skadlig kod"* (fritt översatt från eng., NIST 2013). Ett antivirusprogram söker igenom mjukvara efter både kända virus, genom att matcha mot en signaturdatabas, samt hittills okända virus genom att använda heuristik. Eftersom nya virus täcks in, så kan begreppet självläkande mjukvara ligga ganska nära. Dessutom försöker antivirusprogrammet desinficera (reparera) eller oftare ta bort smittade filer. För att desinficera krävs normalt tillgång till en central databas som innehåller instruktioner om hur programmet bör gå tillväga i det specifika fallet. Detta innebär att antivirusprogram inte är lika självständiga som självläkande mjukvara. Dessutom har de senare ett mycket större fokus på tillgänglighet framför antivirusprogrammens uppdrag att se till att systemets filer inte förändras obehörigen.

I operativsystemet Windows finns mekanismen systemåterställning (eng. system restore) som på vissa sätt är lik SM. Dock agerar den på en mer generell nivå och

inriktar sig på att skydda data snarare än program (kod). Vidare har systemåterställning ett mycket trubbigare tillvägagångsätt när det gäller att avhjälpa problem, eftersom den enda lösningen är att hela operativsystemet återställs, även när felet bara är litet och avgränsat. Av detta skäl kan denna återställande mekanism ta jämförelsevis mycket tid och resurser i anspråk. Dessutom måste användaren (en människa) först upptäcka och kanske diagnostisera problemet samt initiera återställningen, medan SM normalt är autonoma. Slutligen lär sig inte systemåterställning från tidigare erfarenhet utan kommer låta samma problem inträffa nästa gång den specifika situationen inträffar.

Metod

En litteratur sökning gjordes för att identifiera relevanta och intressanta bidrag inom området självläkande mjukvara. Flera välkända databaser genomsöktes: ACM Digital Library, CSA, IEEE Xplore, och Springer. Dessutom användes intressanta referenser från de artiklar som hittats. Särskilt fokus lades på att hitta överblickande forskningsartiklar. För att urskilja de relevanta artiklarna delades de upp mellan författarna som sedan gick igenom dessa enskilt med avstämningar på möten för att upprätthålla gemensamma kriterier för vad som skulle inkluderas. Eftersom rapportens mål främst är utforskande, gjordes ingen strikt systematisk litteraturstudie. Inte heller togs en komplett taxonomi eller kartläggning av området fram. Istället skannades litteraturen mer generellt efter den viktigaste och mest intressanta forskningen. Totalt hittades ett hundratal användbara artiklar, varav den intressantaste tredjedelen presenteras i denna rapport. Halvvägs igenom projektet presenterades dessutom preliminära resultat för intressenter vid en workshop. Detta gav också möjlighet att justera det resterande arbetets inriktning om så behövdes.

Resultat

Keromytis (2007) beskriver den basala livscykeln för självläkande mjukvara. Efter utveckling måste mjukvaran ständigt övervaka sig själv vid körning för eventuella avvikelser och om sådana sker, diagnostisera problemet. Efter diagnostisering måste mjukvaran läka, vilket ofta sker genom att ta fram kandidatlösningar som testas och sedan implementeras i skarpt läge, varvid cykeln börjar om.

Det finns en rad olika typer av mekanismer som kan användas vid de olika livscykelstadierna. Övervakning och diagnostik kan använda sig av en jämförelse med ett känt bra läge såsom en specifikation, eller nyttja en jämförelse med signaturer från tidigare angrepp eller fel, alternativt övervaka datorminnet för att upptäcka avvikande beteende.

Mekanismerna för att åtgärda fel kan i sin tur vara baserade på redundanta komponenter, säkerhetskopior, diversifiering som implementerar samma funktionalitet på olika sätt, periodiska omstarter, eller att stänga inne eventuella problem och därmed hindra dem från att sprida sig, exempelvis genom så kallade virtuella maskiner. Dessutom kan självläkande mjukvara i vissa fall där problemet är mindre allvarligt, besluta att inte försöka åtgärda det utan istället prioritera tillgänglighet till systemet och acceptera ofullständig mjukvara. Vidare kan en del mekanismer användas till både detektering och åtgärd. Ett exempel är att allt som sker i systemet loggas för att senare kunna backas steg för steg när fel inträffar.

En möjlig nackdel med SM är att det inte är en perfekt lösning utan kan riskera att agera mot sådant som inte egentligen är fel (falska positiva) eller missa vissa fel (falska negativa). Att reagera på falska positiva kan leda till att faktiska problem skapas, både genom att systemet slutar fungera som tänkt samt genom att angripare kan få nya vägar in i systemet. Falska negativa däremot kan medföra att legitima uppdateringar inte installeras och ofullständiga ändringar kan försätta systemet i ett instabilt läge. Vidare kräver läkande mekanismer ofta att kod säkerhetskopieras före ändring. Om ändringar då inte utförs fullständigt och säkerhetskopian därmed kvarstår, kan skadlig kod eller fel möjligen överleva genom att gömma sig i kopian.

Självläkande mjukvara har vissa begränsningar. De kan till exempel inte läka den självläkande mekanismen på egen hand, utan kräver då mänsklig inblandning (Frei 2013), även om sådan inblandning kan vara mindre vanligt förekommande och inte så detaljfokuserad som i nuvarande situation utan SM.

En annan nackdel med dessa tekniker är att de kan vara ganska resurskrävande. Detta kan medföra att SM passar sämre på små inbyggda datorsystem, som annars vore särskilt goda kandidater för dessa tekniker som minskar behovet av en mänsklig administratör, som kanske ändå är svår att nå.

Slutsatser

Litteraturgenomgången visade att det, bland artiklarna om självläkande mjukvara, bara finns ett fåtal faktiska lösningar som åtgärdar (läker). Å andra sidan har faserna rörande detektering och diagnostik fått mer fokus. Dessutom finns generella ramverk som beskriver hur SM:s delar hänger ihop samt en del forskning som ser till att självläkande mjukvara inte riskerar att fastna i något av livscykelns skeden. Verifieringsområdet, som kan säkerställa att systemet inte skadar sig självt, är också under uppbyggnad. Detta är av särskilt intresse för verksamhetskritiska system samt industriella styr- och kontrollsystem.

Självläkande mjukvara som inte fungerar som den ska kräver mänsklig assistans, men eftersom denna typ av mjukvara på egen hand hanterar fler och fler fel med tiden, kommer sådan mänsklig inblandning troligen att vara ovanlig. Dessutom

kan självläkande mjukvara minska människans behov av att lösa komplexa problem.

2 Introduction

This section gives the motivation for the work, a historical background, relevant definitions and a brief overview of related fields and concepts.

2.1 Purpose and goal

The purpose of scanning the research frontier is to pin-point new research areas that have a potential for benefitting FOI and its customers. The scanning is part of FOI's strategic renewal.

The report surveys the research areas of self-healing and self-protecting software (SHASP). SHASP are autonomous pieces of software that detect and deal with novel security and safety related problems at runtime. As such, they could provide many benefits to current computer systems. Software in general could be better tested, easier to manage, as well as safer and more secure. The concept of SHASP is ideal for any computer system with a high demand for availability or autonomy such as space and defence applications or critical distributed infrastructure.

In this report we present relevant research found in the field of SHASP. The goal is to explore and provide insight into a research topic that may not have gotten the attention it deserves. This report can therefore provide a suitable starting point for more detailed work that may fill in identified gaps in the literature or new directions that are of interest.

2.2 Background

To help the reader to better understand the origins of the subject, and therefore the subject itself, this subsection details relevant history and biological analogies.

2.2.1 History

The vision of autonomic computing was born from the increased complexity of software systems and a need of a holistic approach. With the advent of widespread web usage, the number of possible sources of errors and the demand for rapid decision making grew very large. Administrators were overworked, systems and software contained too much code to be easily grasped by humans and the possibility of ubiquitous computing systems gave rise to IBM's idea of computers that would fend for themself in an increased fashion (Kephart 2003). An autonomic computing system, as depicted in Figure 1, must know itself, its limitations and its environment. It should also hide its complexity from, and adapt to the user. All this requires the system to configure itself, optimize itself

as well as heal and protect itself. In addition to these four so called self-* (star) properties, many other have been suggested in adjacent fields. Some are parts of the definition of autonomic computing, others are different but definitions vary. It is outside the scope of this report to investigate these issues in depth but suffice to say that dozens of self-* properties were mentioned in ten arbitrary articles related to self-healing and self-protection.

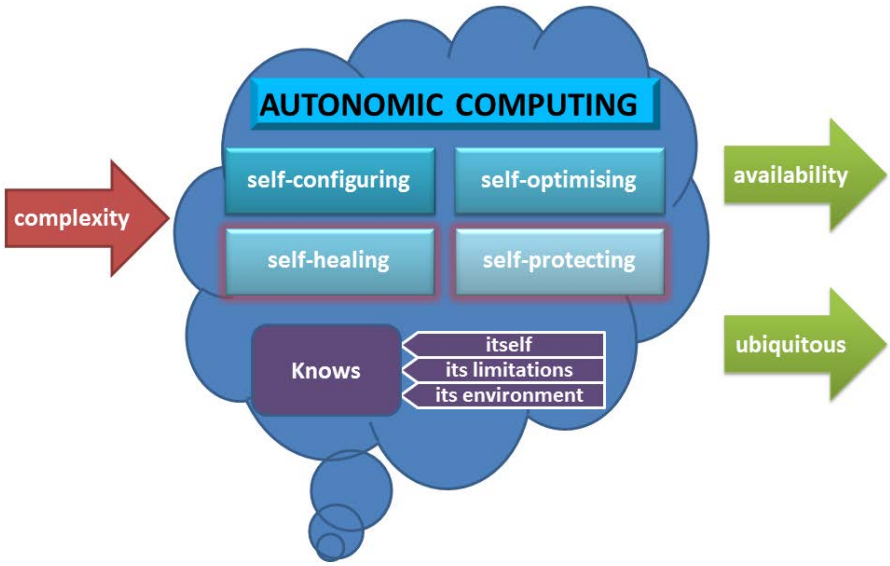

Figure 1 Autonomic computing's parts and advantages

2.2.2 Biology

The original inspiration of self-healing and self-protecting software and autonomic computing as a whole comes from biology and more specifically the human autonomous nervous system with its neurons. It is not entirely obvious how the nervous system corresponds to computer systems and, even if it was not IBM's intention to create a fully artificial intelligence, it may be noted that it is an open question whether this would even be possible.

Perhaps easier to relate to is the human immune system – the concept of viruses and antivirus software is, although perhaps not perfect, an interesting and apparently productive analogy.

The human immune system can be divided into two parts. The first part deals with threats in a rather crude way and consists of, for example, the physical boundary of the skin, the chemicals of the saliva and the gastrointestinal tract and the utilization of temperature coefficients and inflammatory responses. The second part of the immune system consists of more specialised and adaptive

response mechanisms. One group of antibodies diagnoses alien life forms and the havoc they create in the body (Swimmer 2007), while another group reacts and heals (The History of Vaccines 2013). This separation of duties provides a certain protection against so called auto immune responses where the body effectively attacks itself. The protection mechanism prohibits reactions unless the diagnostic antibodies have been involved. The human immune system is also constantly changing. When we are born we inherit a base set of antibodies from our mothers and from that set the immune system is then constantly evolving by learning from inoculation, transfusions and previous attacks.

2.3 Definitions

While we find the entire research field of autonomic computing interesting, this report only deals with the self-healing and self-protection properties and specifically only with the software part of computing systems. Definitions[1] of these concepts vary in the literature, with an early attempt by Kephart (2003):

> *self-healing [software] "automatically detects, diagnoses and repairs problems"*
>
> *self-protecting [software] "automatically anticipates and defends against attacks or cascading failures"*

These definitions are fairly broad and should be considered together with the properties introduced in Figure 2.

Other attempts to define self-healing definitions from the literature are given below:

- "a self-healing software system is a software architecture that enables the continuous and automatic monitoring, diagnosis, and remediation of software faults" (Keromytis 2007)
- "a [self-healing] system will detect, diagnose, and repair performance problems and hardware/software faults automatically" (Duan 2009)
- "Self-healing is a bottom-up approach, where the components of the system heal the damage from inside" (Frei 2013)

[1] We decided on a single definition in Swedish covering both self-healing and self-protection: *"När självläkande mjukvara angrips eller drabbas av såväl nya som gamla fel, kan den autonomt och dynamiskt (genom realtidsbeslut) upptäcka samt diagnostisera problemet och läka de skador som uppstått."*

- "[self-healing is] the property that enables a system to perceive that it is not operating correctly and, without (or with) human intervention, make the necessary adjustments to restore itself to normalcy." (Ghosh 2007)
- "Self-healing means that an ACS [Autonomic Computer System] must detect failed components, eliminate it, or replace it with another component without disrupting the system." (Nami 2007)

There are also alternative definitions of self-protection:

- "[Achieveing self-protection means that] an ACS [Autonomic Computer System] must identify and detect attacks and cover all aspects of system security at different levels such as the platform, operating system, applications, etc. It must also predict problems based on sensor reports and attempt to avoid them. " (Nami 2007)
- "[self-protecting systems are] systems capable of detecting and mitigating security threats at runtime" (Yuan 2010)

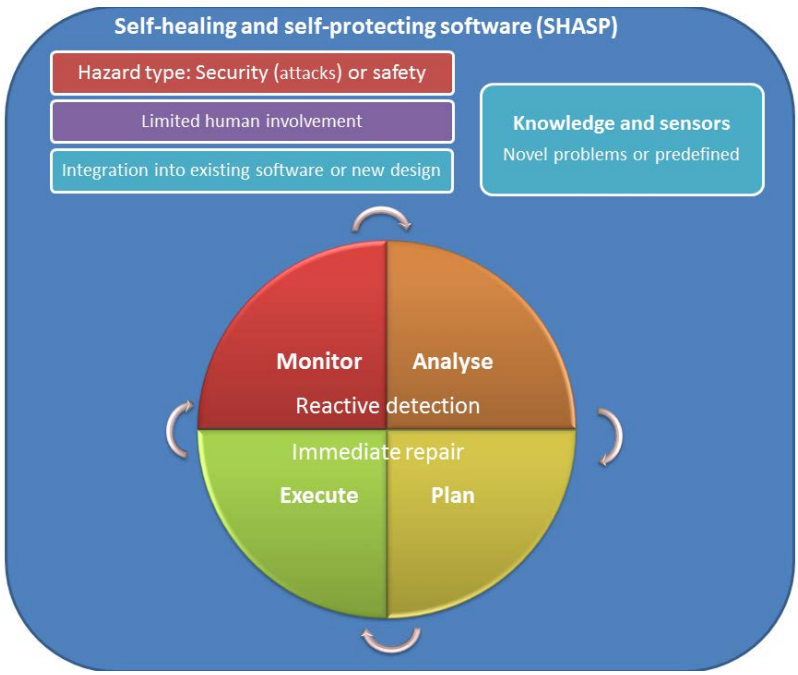

Figure 2 – The properties of self-healing and self-protecting software

The self-healing techniques deal with software or hardware faults and are focused on continued execution and crash avoidance. Self-protection techniques

handle defensive measures against antagonistic attacks and try to limit the spread of an attack, although the removal and reparation of damages after an attack is often left to other mechanisms. In each case the focus is on handling unknown (new) faults or attacks.

This report covers both self-healing and self-protection techniques, but it also reflects that the remediating stage of self-healing software techniques is more prominent than the one in self-protection.

Figure 2 shows some of the properties and classes of self-healing and self-protecting software. Note first of all the four lifecycle stages; monitor, analyse (a discovered problem), plan and execute (a remediating action). One distinctive property of SHASP techniques is that they deal primarily with new and unknown safety or security problems (i.e. faults or attacks), although as it can be seen in Table 1, exceptions do exist. Since the problems are new, and above all because the software needs to investigate itself, proactive measures must be eschewed in favour of reactive ones. Furthermore, these problems are taken care of by a partially or wholly autonomous and dynamic system where healing decisions are made with only limited human involvement and without having to first shut down the running component in question. This is particularly useful for real-time systems where availability trumps everything else. Additionally, SHASP repairs components instead of simply replacing them with identical backups. This is especially important for the ability to handle unknown problems. Finally, the defensive techniques are typically integrated into existing systems rather than at the original design phase, but this also puts some limitations on the precision of the mechanisms.

These mechanisms are not intended to be used as an answer to all security and safety issues in software, nor to replace existing solutions, but to complement them and act as a last resort in case everything else has failed. As mentioned above, human intervention is usually not completely eliminated, only reduced. Also, many of the systems suggested in the literature will need a learning phase to establish how to best deal with novel threats and therefore such techniques perform better and better with time. Many threats have novel specifics, but also many things in common with previous, known, threats. By placing threats in different classes, SHASP can react more efficiently. For instance, while two buffer overflow attacks may be different in execution, they have the same general goal and can be remedied by the same bounds checking or executable space protection.

2.4 Related concepts

As previously mentioned, this report is concerned only with two parts of the original idea of autonomic computing. Although both self-healing and self-protection have already been defined in this report, it may be of use to contrast

these concepts with other similar kinds of software defence mechanisms. For this reason, this part of the report will briefly mention some of these adjacent concepts. Please note that while each concept has a slightly different focus, it is difficult to draw any firm lines between them and often they overlap somewhat.

2.4.1 Fault tolerance

Fault-tolerance is concerned with *both* soft- and hardware faults and the main goal is to continue functioning in the presence of faults. Furthermore, it is typically oblivious of the reason for the fault and its focus does not usually include security but is limited to safety. Not all fault-tolerant techniques are self-healing (e.g. redundancy), while all self-healing techniques are fault-tolerant.

2.4.2 Intrusion tolerant systems

These types of systems are able to "prevent the intrusion from generating a system failure" (Veríssimo 2002). Also, intrusion detection and prevention systems "detect suspicious events when they happen and inform the system manager" (Gollmann 1999) while intrusion response systems "react immediately to security alarms by taking appropriate actions" (Gollmann 1999). All these systems can be considered part of self-protecting software if they are autonomous and dynamic (take decisions dynamically at runtime). However, Yuan et al. (2010) considers that intrusion tolerant systems and intrusion reaction systems "are still intrusion-centric and perimeter based and as such do not yet constitute true self-protection".

2.4.3 Recovery-oriented computing

Recovery-oriented computing (ROC) "takes the perspective that hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved, […] and fast recovery is how we cope with these inevitable errors" (Patterson 2002). ROC considers that administrators are necessary to recover fast from failures and that it is better to give them more useful tools to help them do so, instead of focusing on full automation. Both ROC and SHASP are approaches aiming to achieve highly available computing.

2.4.4 Self-repair

Another part of autonomic computing is self-repair, which is defined by Frei et al. (2013) as separate from self-healing, while acknowledging that the terms are often used interchangeably in the literature with varying definitions. The paper goes on to describe self-healing as "a bottom-up approach, where the components of the system heal the damage from inside", compared to self-

repair's "top-down approach, where the system is able to maintain or repair itself". The authors give an example related to the human body to better understand the difference: a broken fingernail will just grow back by itself and the broken part will be removed without any conscious decision from the brain, this is therefore self-healing. If the person with a broken nail instead uses its other hand to file the broken nail using a conscious decision of the brain, then self-repair instead applies (Frei 2013). However, given that both self- properties are "conscious" and the distinction between inside and outside is fuzzy, this analogy is not altogether very clear. Furthermore, it should be noted that self-healing is not as bottom-up as fixing each specific fault at machine code level, instead it deals with fault classes and stresses availability.

2.4.5 Antivirus software

Another concept related to those of self-healing and self-protection is that of antivirus software. These well-known applications "monitor a computer or network to identify all major types of malware and prevent or contain malware incidents" (NIST 2013). An antivirus program both checks after known viruses by using a signature database and utilises heuristics to find new viruses. As it tries to detect new viruses, it can be seen as a self-protecting system. Furthermore, if a virus is found, then the antivirus software will either delete the infected file or try to disinfect it, where deletion happens more often than disinfection. When disinfecting files, the antivirus program will typically send the virus signature to a central database and receive directions on how the virus can be removed from the infected file. Although some healing of the infected file takes place, antivirus software cannot really be considered a self-healing system since there is no dynamic and autonomous decision making on how the program should be healed. Nevertheless in the cases where such decision making is present, the antivirus system may be considered both a self-protecting and a self-healing system. However, antivirus software places a far greater emphasis on data integrity than SHASP's availability – the latter software sometimes refrain from remediation in case the problem does not seriously affect core function.

2.4.6 System restore

The Windows operating system has a special mechanism which in some way is similar to SHASP. However, it acts on a more abstract level and targets saving data instead of focusing on programs (code). Furthermore, it takes a much more blunt approach to remediation, where the entire operating system is restored even in the case of only one small and specific fault. For this reason, the healing mechanism may be rather time-consuming and otherwise resource intensive. Additionally, the user (human) must first detect and possibly diagnose the problem and then initiate the remediating process, while SHASP is typically automated. Finally, system restore does not learn from previous executions and

will allow the same fault to be introduced next time the specific circumstances occur.

3 Method

We have performed a literature search to identify important and interesting research in the area of SHASP. As this is primarily an exploratory study, we did not conduct a strict systematic literature review nor do we present a taxonomy or survey of the field. Several well-known databases were used to search for relevant literature: ACM Digital Library, CSA, IEEE Xplore, ScienceDirect and Springer. The initial search query used was as follows:

("self-healing" OR "self healing" OR "self-repairing" OR "self repairing") AND ("software" OR "computing").

Different combinations and settings were used for different databases. Also in order to find relevant *surveys* and *reviews*, some extra search queries were made including these words. Furthermore, some literature was discovered by going through interesting references of selected papers as well as through general Google searches. Additionally, some further searches were made for *self-protection* when this was discovered to be a relevant keyword, although this did only generate a few new articles. Not all papers were available for free in their entirety and hence they were left out, leaving us with about 90 usable articles.

The articles were put through a screening process as to only keep those deemed relevant to this literature review. For this task the articles were divided between the three authors and the screening was performed individually, by each author. Regular meetings were utilised to discuss what was relevant to this study and to maintain joint inclusion criteria. As a result of the screenings and meeting discussions we feel confident that the selection presented in this study includes some of the most relevant freely available publications within the area, while not having any requirement for completeness.

The selected articles were examined in detail to establish not only if each article truly was relevant to our study, but also to understand what problem it tried to solve, what approach it took in solving the problem and whether any empirical validation such as a proof-of-concept or simulation was performed.

Halfway through the project, preliminary results were presented to stakeholders at a workshop. This also served to adjust the direction for the remainder of the work.

4 Results

This section describes the results of the literature review. A general theoretical description is followed by a summary of articles with both concrete and abstract examples of SHASP as well as a classification of these articles. Finally the section directs the reader to surveys that may provide further insight into the matter at hand.

4.1 General theory

Keromytis (2007) describes the basic life cycle of self-healing software. After deployment, a piece of software must continually monitor itself for anomalous events in which case it proceeds with self-diagnosis. After a fault has been identified the software must self-adapt to allow for candidate fixes to be generated. Finally, the candidate fixes are automatically tested and deployed, after which the cycle starts anew. Instead of self-healing's detection-diagnosis-remediation, self-protecting software uses the slightly different notions of monitoring, analysing, planning and executing. However this does not constitute a major difference and the terms are sometimes used interchangeably (as can be seen in Figure 2).

Many different kinds of mechanisms may be utilised for the different stages of self-healing and self-protecting software.

Detection and diagnosis may rely on comparisons with a known good state such as a hash or specification, comparisons with signatures from previously known attacks or faults, or monitoring of for example the memory to discover irregular behaviour.

Casanova et al. (2011) detect faults and investigate possible causes of these. Trace abstractions describe the fault while fault candidates are ranked by various quality attributes (e.g. performance). This provides general fault diagnosis, with the exact nature of the fault being irrelevant. To be able to detect and handle faults, computations are monitored using certain criteria and transactions are made atomic. This approach may be combined with black-box components.

The mechanisms for remedying may in turn be based on redundancy and backup, diversifying by implementing the same functionality in different ways, periodic reboots (i.e. rejuvenation) or containment with sandboxes, virtual machines and proxies. In some cases with less worrisome threats, SHASP will make the decision to continue operation rather than accepting any downtime for remediation.

Some mechanisms contribute to both detection and remedying. These include byzantine voting as well as logging with snapshots or transactions together with rollbacks.

SHASP are not the be-all-end-all solution to security and safety issues in software. Apart from the difficulty of automatically handling novel problems at runtime, they also need to work together with the rest of the system. In some cases integration of the relevant techniques into existing systems may be feasible, in others – especially when source code is not openly available or files are encrypted when not in use – a complete redesign may be necessary. Furthermore, false positives – when the healing and protection mechanisms are invoked erroneously – may be a significant challenge (Swimmer 2007). This may be of particular concern in critical systems where formal verification is typically applied before use. Indeed, remedying mechanisms may present an attacker with a new way of altering the system software. Conversely, false negatives – where code is not changed when it would be beneficial to do so – may cause legitimate system updates to fail to be applied. The system may also reach an unstable state in case changes are incomplete (c.f. Windows system restore). Additionally, healing and protection mechanisms typically require some code or state to be saved as a backup during program execution. It may be a concern that malicious code may in some instances be able to survive cleaning processes by hiding in such backups.

The concept of SHASP in itself also has certain limitations. For instance, always finding the root cause of a failure is a very difficult proposition (Gao 2003). Furthermore, a failing healing mechanism will need human assistance as it cannot heal itself (Frei 2013) and it may be more difficult for a human to understand the system when certain complexity has been hidden. However, it is possible that such human involvement will be less common and less detail-oriented than in the present case.

Another drawback of these techniques is that they typically generate a certain amount of overhead. This may cause problems on systems with limited resources. This is likely of particular importance for small embedded systems, which would otherwise be especially good candidates for these techniques that alleviate the need of a human administrator. However, O'Sullivan (2011) have shown that overhead can be kept at a very low level, at least for self-protecting software regarding low-level software attacks.

4.2 Summary of selected articles

In this section we present the most interesting and relevant articles. We categorise them according to their approach to self-healing/self-protection, either how they heal/protect or what they heal/protect. The last two categories are an exception from this classification, one of them presents some approaches to

verification of self-healing and self-protecting systems and the other presents a more general framework for self-adaptation which can be used in systems with self-* properties.

4.2.1 Self-healing by error virtualization

The idea behind error virtualization is that portions of an application (for example each function execution) can be treated as a transaction. If a transaction experiences a fault or if a vulnerability is exploited, the transaction is aborted. Since the program state is saved before each transaction starts to execute, the program state can be restored to what it was before the transaction. Saving program state introduces overhead, and there are different approaches to how to reduce this overhead.

Error virtualization is not intended to replace patching the software; instead it is intended to provide a temporary fix until the vendor provides a patch. Some examples of error virtualization solutions are given below.

Sidiroglou et al. (2005a) present a reactive system for handling a variety of software errors, like remotely exploitable vulnerabilities or bugs that cause abnormal program termination. The reactive system tries to protect itself against recurring faults. The main idea is that, the first time a fault happens, some application monitors are able to locate the error or attack and mark the affected sections for emulated execution. The application is restarted and the marked sections are run in an emulator, the program state is snapshotted and all affected instructions are executed on a virtual processor. If all the code is executed without problems, then the virtual processor copies its internal state to the real CPU. If the emulator detects that a fault is about to happen, then the excursion is aborted and the state is restored to the one in the beginning of the emulation. Then the function containing the fault is forced to return an error value. In this way errors that were not actually considered in advance of execution may be taken care of. It is not always straightforward what error to return, but the paper considers some simple heuristic for this (for example if the return type is an *int*, then the function returns -1).

The solution considers a server-application and deals with errors that are caused by one input, not a combination of inputs. It can only deal with failures that can be algorithmically determined, meaning that some other code can determine where the fault is located and what kind of fault it is.

The paper also includes a proof-of-concept where the solution is tested on real applications such as Apache, sshd and Bind. As only the affected sections are run in the emulator, the performance overhead due to saving program states is kept low. The goal is to avoid recurring errors, while an error has to happen at least

once so that the monitors can detect it and find out where in the code it happened. The self-healing part is that the function where the failure occurred is aborted with an error and the application does not crash.

Another solution based on the same idea of error virtualization is proposed by Sidiroglou et al. (2005b) and focuses on buffer overflow attacks. The basic idea is that a buffer overflow or underflow causes an exception which is caught and the program recovers execution from some suitable location. The assumption is that the program can handle truncated data in a buffer. In order to monitor buffer overflow attacks, the static buffers are moved to the heap, by dynamically allocating the buffer when entering a function and two extra read only memory pages are allocated that surround the desired memory region. Also, a signal handle is inserted that prints the call stack and the name of the buffer that is overflowed. The rest of the solution follows the same line of action as in (Sidiroglou 2005a): the function is monitored and if a problem (i.e. buffer overflow) occurs, the function is aborted and some error is returned.

The notion of rescue points in the context of error virtualization was introduced by Sidiroglou et al. (2009). The rescue points recover software from unknown faults while preserving both system integrity and availability. Rescue points are "positions in existing application code for handling programmer-anticipated failures which are automatically repurposed and tested for safely enabling general fault recovery". When an error occurs at some arbitrary location, the proposed system called ASSURE restores execution to the closest rescue point where there is a good chance that the error can be handled correctly. ASSURE uses quality assurance testing techniques to generate bad inputs to an application and identify candidate rescue points. After a failure occurred for the first time, a replica of the application is used to investigate what rescue point should be used in future program executions. The application is therefore dynamically patched to self-checkpoint at the rescue point.

The solution handles unexpected failures, caused by bugs and there is no need for source code (a typical goal of many solutions in the literature). The solution is targeted at server applications and the authors provide a proof-of-concept.

For the interested reader, other solutions based on rescue points have been proposed by Portokalidis and Keromytis (2011) as well as Zavou et al. (2012).

4.2.2 Self-healing by error patching

Perkins et al. (2009) propose a system for automatically patching errors in deployed software. The system observes the behaviour of the running code and tries to infer invariants, or properties, that characterize normal execution. The invariants can be values of registers or memory locations, or they can be control

flows. The invariants are always satisfied during normal execution, and, when the system observes abnormal execution, it tries to patch the system by changing the values of the invariants – not the actual code. The patches are not guaranteed to work; it is a learning cycle and each patch is evaluated to establish how well it works. The system tries different patches for a given error and, with time, learns to better choose a patch. Note that the system cannot detect all failures, but only the ones for which it has a detection monitor.

The paper also provides an implementation of the system and a red team evaluation.

4.2.3 Self-healing for workload balancing

Duan et al. (2009) present a solution for workload balancing when workload changes cause failures (such as performance problems) in database-backed web services. The authors state that there are many mechanisms that could work for a self-healing database system; the problem is that there is a lack of suitable policies to invoke these mechanisms automatically, efficiently and correctly in case of failure. The policies should detect failures in a timely fashion, have a fix attached and a right time to apply the fix. The failures dealt with seem to be easy to detect but hard to remedy. A failure is solved by applying a fix: a resource-based fix (i.e. allocation of physical resources like memory or CPU) or a configuration-based fix (like changes to the design of the database). Each fix is assigned a cost and the one with the least cost is chosen. The cost is given by the time taken to get back to a healthy state and the amount of resources needed. Performance and cost models are learned from offline experimentation. The solution is a combination of proactive and reactive approaches.

4.2.4 The danger model for self-protection and self-healing

Swimmer (2007) focuses on protecting an internal network from external attacks, especially worm attacks. The goal is to use the so-called danger model to protect against new unknown worms and to limit their spread as much as possible.

The danger model is a fairly new theory on how the human immune system works. The old theory relied on the idea of “self” and “non-self” (SNS), and that the immune system could rely on SNS to find the foreign body and establish a memory of past infections. The danger model does not entirely dismiss the SNS theory, but it also relies on danger signals from injured cells. There signals are secretions from cells when they die from external causes.

Using ideas from the danger model, the Autonomic Defense Network proposed in the paper includes the following elements:

- Danger sensors, which are attached to applications and monitor for signs of distress. They also extract attack signatures when an attack occurred.
- Program filters, which scan input to programs for signs of malicious content. The filters use attack signatures.
- Network monitors, in form of network IDSs or Honeypots are used to monitor the network using attacks signatures.
- Chokes are used at router or firewall level, to limit the spreading of attacks by suppressing traffic.

The different elements work together to protect against unknown attacks.

The danger sensors have a very important role in the system and it is important that they are well-developed. An example of a danger model is to monitor the system calls and decide if this constitutes normal behavior. For this decision, static analyses of the binary executable are performed and a non-deterministic finite automaton (NFA) is created. The NFA includes all the possible sequences of system calls. When a system call is detected that does not fit any of the possible sequences of system calls, then an alert is triggered.

Elsadig and Abdullag (2009) propose a model for intrusion prevention and self-healing for network security based on the danger theory of the human immune system. The paper focuses on the intrusion prevention part and uses a multi-agent based approach, each agent being a system component that has its own goal. Each agent's function and structural specifications are detailed and grouped into sets of roles. The abstract architecture is modeled as a discrete-event system using Petri nets. Deadlock avoidance in the multi-agent system is considered as an initial key property, and it is evaluated using the liveness and boundedness properties of the Petri net model. While the paper does not propose a strategy for self-healing as such, it provides a framework that integrates self-healing and intrusion prevention into the same system.

4.2.5 Self-healing by redundancy

Perino (2013) proposes the design of a general self-healing framework to deal with functional failures at runtime in component-based applications. It includes detecting failures, recovering the system state and assuring state consistency, and finally healing the fault. The approach taken to heal is to exploit the available redundancy of components. To identify if there is any exploitable redundancy, the author sifted through Java libraries for operations that are equivalent, in the sense that they lead to the same result. The system state is saved at several points of the execution and then, when a failure is detected, the system is rolled back to

one of the saved points. Then the failing operation is replaced by an equivalent operation.

4.2.6 Consequence-oriented self-healing

Dai et al. (2011) propose a new approach that is not based on finding and fixing bugs in the software, but is instead based on the idea of consequences of software bugs and eliminating those consequences. The motivation behind this approach is that testing and fixing bugs would require stopping and recompiling the software, which is impractical in real-time applications. Also, current fault detection and isolation tools can only identify a bug at the module or component level, but for remediating a bug, the level of class, object, function and even the specific line of code would need to be identified.

The main idea is that, for continuing to run the system, it is enough to identify the consequences of the existing bugs from the symptoms and to remedy those consequences before a catastrophic failure occurs. One advantage, according to the authors, is that different causes can have the same consequence; therefore the number of consequences may be smaller than the number of causes. Also most prescriptions can be straightforward and general.

A simple example given by the authors is a memory leak that could be caused by forgetting to release memory when objects are deleted, which in the long run could lead to memory exhaustion. Although the bug may exist in arbitrary parts of the program, the consequence is the same: memory consumption. The approach presented focuses on remediating the consequence – i.e. in this example reclaiming the unreachable memory without stopping the running process or rebooting the system.

4.2.7 Self-healing for COTS component integration

Chang et al. (2009) present a strategy to heal common integration problems due to wrong usage of interfaces when integrating COTS components into software systems. The motivation behind this work is that common integration problems and their fixes are usually well-documented but nonetheless commonly occur because application developers have a good understanding only of their own application and not of the COTS parts.

The proposed strategy is to allow COTS developers to develop healing connectors for common misuse of their COTS components. Healing connectors are software modules that can be injected into software applications that integrate COTS components. A healing connector is activated by a raised exception and it includes a connector that intercepts exceptions, a set of healing strategies and a specification of the points where the connector must be injected. In order to

identify if a healing strategy is available for the given state, the healing connector inspects the current state of the system to see if a consequence of a problem that can be healed was found. Note that the healing connector comes with a set of predefined healing strategies and that there is no learning of new consequences or trying of new healing strategies. If a healing strategy has been identified for the problem, the connector tries to heal the system. If the attempt does not succeed the exception is re-thrown.

In order to improve the software system over time, the activities of the healing connectors are logged so that software developers can inspect the used healing strategies and permanently fix the problems if possible.

A simple example is an application developer who forgets to invoke COTS methods in the appropriate order.

4.2.8 Control-based self-healing with a supervisor module

FastFIX (Gaudin 2011) is a project that aims to provide improved remote maintenance of software including some self-healing properties. The project has a control-based approach to self-healing where a supervisor module monitors the application and only allows desired behavior. In order to accomplish this, a pre-deployment and a post-deployment phase are needed.

The pre-deployment phase includes code instrumentation and model extraction. Model extraction is performed in order to obtain a model of the behaviour of the system which is used in conjunction with a set of desired behaviours. Code instrumentation is performed to embed the supervisor and to introduce observation and check points.

The system is controlled at runtime by the supervisor module. The sequence of method calls that occur at runtime is considered and the supervisor has to sign off on each method before its use. The authors give an example of how this can be accomplished. Nevertheless, designing a supervisor is a challenging task since the high complexity of software makes it difficult to take into account all possible failures that can occur. Therefore the supervisor might need to adapt to newly observed undesired behavior at runtime.

The self-healing approach presented in this paper relies on control theory: a supervisor module controls the behavior of the system at runtime and it is able to adapt itself to previously unknown behaviour.

4.2.9 Self-healing by runtime execution profiling

Fuad et al. (2011) propose a self-healing scheme by runtime execution profiling. The solution only considers transient faults like network outage, disk space outage and memory overload and does not deal with non-transient faults caused by bugs. A technique is presented that can choose self-healing actions for an unknown fault scenario based on matching the scenario to already established common fault models, which are updated over time as new faults arise. The fault models consist of signature traces (ST) and possible fixes for a specific fault scenario. Over time, more and more STs are gathered.

When a fault occurs, the failure ST is compared to common fault model STs. The most appropriate candidate fix – as calculated by a matching algorithm – is applied. If a fix takes the system to a stable state then the fault has been fixed, the common fault model is updated and the process can continue. If the fix does not work, then the next possible candidate is considered. If no known fix can take the system into a stable state, then the human administrator is notified. After a successful fault fix, the fault model is updated with the new fault and the attached ST.

The goal of this proposal is not to completely exclude the administrator, instead it is to reduce the number of times the administrator is involved. Also, the proposal automates the finding and matching of new faults with known faults, while the fixes to the faults still have to be specified by the administrator.

In order to run this on software systems, the code needs to be transformed and injection techniques are needed to assure that the code can be monitored and also that the matching algorithms are triggered. The authors have already proposed solutions for that in a previous work (Fuad 2008).

4.2.10 A framework for runtime adaptation for SHASP

A framework for runtime adaptation is proposed by Griffith and Kaiser (2006), which is necessary in autonomic computing. For example, when self-healing is considered, adaptation is needed to perform diagnosis and to remediate faults. In this paper, the adaptation problem is considered for fine-grained adaptation (e.g. restarting and refreshing individual components or sub-systems) and with the adaptation mechanism separate from the main software, allowing for adaptation where no source code is available.

The article presents solutions for both managed (e.g. an application written in Java) and unmanaged applications (e.g. an application written in C). The feasibility of performing adaptation using Kheiron/C is demonstrated.

4.2.11 Verification of self-healing and self-protection software

This subsection has a slightly different viewpoint than the previous ones since it deals with quality assurance aspects of SHASP rather than its construction or usage in general.

Eckardt et al. (2013) consider a component based architecture in a system with self-* properties. The software in such systems needs to be configured to satisfy the self-* properties, which can mean adding or removing components and adding or removing component interaction. The state space of such a dynamic system becomes so complex that current verification approaches like model checking or theorem proving do not scale. The article proposes a new architecture where the construction and reconfiguration of the architecture can be controlled by using graph transformation rules and, in such an architecture, the verification of safety and liveness properties only has to be carried out for an initial state instead of the whole system.

Another verification approach is taken by Bucchiarone et al. (2009).
The goal of the article is to formalize self-repairing (or -healing) Dynamic Software Architectures (DSAs) in order to verify correctness and completeness of the self-repairing specification. A DSA is used in systems where system evolution is necessary and it is argued in the paper that all modern software systems need it since the requirements and the context evolve over time. A specification is defined as complete if "each desirable configuration different from [the initial configuration] can be reached by applying repairing mechanisms" and it is correct if "there exist repairing productions for each reachable configuration that does not belong to the set of the desirable configurations".

4.3 Classification

Table 1 presents an overview of the papers presented in this section and their approach to self-healing and self-protection, reflecting to what degree they fulfil the idea of autonomic computing. We include only those articles whose solutions can be classified according to the different characteristics from Figure 2. As such, the article by Elsadig and Abdullag (2009) is left out since it deals with the integration of pre-existing self-protection solutions with other security solutions and not the actual design of self-protecting software. Eckardt et al. (2013) and Bucchiarone et al. (2009) are also not included due to their general nature and pre-supposing of the existence of a self-healing solution.

The table's first column indicates the article, while the second reflects whether the article concerns itself with self-protection (security) or self-healing (safety). The following column describes if the article solution uses a human

administrator or if it is fully automated (auto). The fourth column provides details on whether the article's solution supports integrating into already existing software or if the desired degree of self-protection or self-healing must be considered already at the software design phase. The next column describes the possibility for the mechanism in question to handle novel threats or only those that have been defined in advance. A final column indicates if the suggested solution does all its work during execution of the main (business-beneficial) program or if some proactive effort is also carried out.

Table 1 Overview of the articles and their properties

property / **article**	Security or safety	Auto or manual	Integration or new development	Novel threats or predefined	Proactive or reactive
Casanova 2011	Safety	Auto	Integration	Predefined	Reactive
Sidiroglou 2005a	Both	Auto	Integration	Predefined	Reactive
Sidiroglou 2005b	Security	Auto	Integration	Predefined	Reactive
Sidiroglou 2009	Safety	Auto	Integration	Novel	Reactive
Perkins 2009	Safety	Auto	Integration	Novel	Reactive
Duan 2009	Safety	Auto	Integration	Novel	Both
Swimmer 2007	Security	Auto	Integration	Predefined	Both
Perino 2013	Security	Auto	Integration	Predefined	Reactive
Dai 2011	Safety	Auto	Integration	Novel diagnosis, but predefined fix	Reactive
Chang 2009	Safety	Auto, with manual backup	At development	Predefined	N/A
Gaudin 2011	Safety	Auto	At development	Both	Reactive

property / article	Security or safety	Auto or manual	Integration or new development	Novel threats or predefined	Proactive or reactive
Fuad 2011	Safety	Auto, with manual backup	Integration	Novel diagnosis, but predefined fix	Reactive
Griffith 2006	Both	N/A	Integration	N/A	Reactive

4.4 Surveys

For readers that would like to read more about SHASP, two surveys that work well as gateways to additional information are detailed below.

In Psaier and Dustdar (2011) a comprehensive survey of existing self-healing techniques is presented. The paper is also a good introduction to the area of self-healing systems in general as it discusses the background and principles of self-healing. It also classifies the solutions by area of research: e.g. embedded systems, operating systems, architecture based.

The paper concludes that a system with self-healing properties can be identified as a system that comprises fault-tolerant, self-stabilizing and survivable system capabilities and, if needed, is human supported. A self-stabilizing system is a system which will end up in a correct state after a finite number of execution steps, no matter what the initial state is. Survivability is “the ability of a given system with a given intended usage to provide a pre-specified minimum level of service in the event of one or more pre-specified threats” (Westmark 2004).

According to the paper, the commonly used recovery techniques are: replacement, balancing, isolation, persistence, redirection, relocation and diversity.

Yuan et al. (2010) survey self-protection mechanisms using the definition “security threats are detected and mitigated at runtime, both a ‘reactive’ perspective, the system automatically defends against malicious attacks or cascading failures, and a ‘proactive’ perspective, the system anticipates security problems in the future and takes steps to mitigate them”.

Motivating examples of external and internal threats are given as well as a list of related taxonomies. The paper looks both at the objectives and intent of research of self-protection and at how self-protection can be achieved. The authors also present a list of research directions that has been constructed from the gaps found in the studied literature.

5 Conclusions

In this report we have presented a literature review of self-healing and self-protecting software (SHASP). The goal of the review was to get an understanding of the maturity of the research field and find possible uses for the techniques and mechanisms discovered. A self-healing system *"automatically detects, diagnoses and repairs problems"*, while a self-protecting system *"automatically anticipates and defends against attacks or cascading failures"* (Kephart 2003). Both techniques prioritize availability and therefore have a focus on continuing execution despite the presence of problems.

The self-healing and self-protecting systems found in the literature try to solve a variety of problems by using a number of different solutions. The most common problems to solve are non-transient faults (i.e. bugs) and network attacks. Some other problems to solve are transient faults (i.e. network outage), integration concerns and performance issues. Many of the presented solutions try to fix new and unknown problems or to protect from unknown attacks, but there is never a guarantee that a good fix will be found.

In terms of actual solutions for the remedying healing mechanism the area is not very mature. There are just a few examples where healing actually takes place as an autonomic and dynamic activity. Nevertheless, the parts of self-healing techniques that are related to detecting and diagnosing faults have been more extensively studied in the literature. Also there are general frameworks for how to fit all the parts together; the detection, diagnosing and healing. Some steps have also been taken towards verifying properties such as liveness of self-healing systems. Verification is of particular interest when using SHASP in mission-critical systems. For now these verification techniques can only be applied to specific solutions.

Since no in-use software system is bug-free and no security solution can guarantee all-encompassing protection, we believe that any software system could benefit from the use of self-healing and self-protecting techniques, as long as the introduced overhead is not too costly. We have not tried to find detailed figures on what amount of overhead each article's solution introduces, but this is an issue that needs to receive further attention as the field matures. Systems, such as space probes or unmanned vehicles, that need to operate independently from administrators, may be an especially good fit for self-healing and self-protecting software. For instance, Stroupe (2002) project the use of autonomous deep space exploration within the next ten years, while Kavulya (2011) mention that some hardware redundancy has already been replaced by software techniques. However, this puts further limits as to acceptable overhead. This issue is also an important one for potential use in high-availability (real-time) systems. In both cases, a heftier price tag due to higher performance requirements may be more tolerable than for typical office applications.

SHASP have some inherent or potential drawbacks and limitations. Since the current state of research is still quite immature in general, this specific topic has not received much attention from the research community so far. However, it is well-known that designing a general and automatic solution to a group of problems is far more difficult than solving just one of those problems. In this way, self-healing and self-protecting software constitute one degree of complexity higher than the traditional ways of dealing with bugs and attacks. A reaction that is not carefully measured or erroneously invoked by false positives, will result in incorrect and potentially malicious changes to code. Furthermore, false negatives may lead to issues that linger instead of being dealt with. Malicious code may also be able to hide in code batches that are backed up before remediation and in this way potentially survive longer than otherwise.

A failing healing mechanism will need human assistance, but as these mechanisms autonomically deal with more and more faults with time, such human involvement will likely be rare. Furthermore, self-healing and self-protecting systems may alleviate human's need of solving complex problems.

6 References

Bucchiarone, A., Pelliccione, P., Vattani, C. & Runge, O. (2009). *Self-Repairing systems modeling and verification using* AGG. Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. pages181-190.

Casanova, P., Schmerl,B., Garlan, D. & Abreu,R. (2011). *Architecture-based run-time fault diagnosis*. In Proceedings of the 5th European conference on Software architecture (ECSA'11), Ivica Crnkovic, Volker Gruhn, and Matthias Book (Eds.). Springer-Verlag, Berlin, Heidelberg, pages 261-277.

Chang, H., Mariani, L. & Pezze. M. (2009). *In-field healing of integration problems with COTS components*. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, pages 166-176.

Dai, Y., Xiang, Y., Li, Y., Xing, L. & Zhang, G. (2011). *Consequence Oriented Self-Healing and Autonomous Diagnosis for Highly Reliable Systems and Software.* IEEE Transactions on Reliability, vol.60, no.2, pages 369 - 380.

Duan, S., Franklin, P., Thummala, V., Dongdong, Z. & Babu, S. (2009). *Shaman: A Self-Healing Database System*. IEEE 25th International Conference on Data Engineering (ICDE '09). Pages 1539-1542.

Eckardt, T., Heinzemann, C., Henkler, S., Hirsch, M., Priesterjahn, C., & Schäfer W. (2013). *Modeling and verifying dynamic communication structures based on graph transformations*. Computer Science - Research and Development. Volume 28, issue 1, pages 3-22.

Elsadig, M. & Abdullah, A. (2009). *Biological Inspired Intrusion Prevention and Self-Healing System for Network Security Based on Danger Theory*, Computer Science Letters, Vol 1, No 2.

Frei, R., McWilliam, R., Derrick, B., Purvis, A., Tiwari, A. & Di Marzo Serugendo, G. (2013). *Self-healing and self-repairing technologies*. International Journal of Advanced Manufacturing Technology, 69(5), pages 1033–1061.

Fuad, M.M. (2008). *Code Transformation Techniques and Management Architecture for Self manageable Distributed Applications*. Twentieth International Conference on Software Engineering and Knowledge Engineering, USA.

Fuad, M.M., Deb, D. & Baek J. (2011). *Self-Healing by Means of Runtime Execution Profiling*. Proceedings of 2011 International Conference on Computer and Information Technology (ICCIT 2011), pages 202–207, Dhaka, Bangladesh.

Gao, J., Kar, G. & Kermani, P. (2003). *Approaches to Building Self-Healing Systems Using Dependency Analysis.* IBM Research Report, IBM Research Division.

Gaudin, B. & Hinchey, M. (2011). *FastFIX: An approach to self-healing.* Federated Conference on Computer Science and Information Systems (FedCSIS), pages 957-964.

Ghosh, D., Sharman, R., Rao, H.R. & Upadhyaya, S. (2007). *Self-healing systems - survey and synthesis*. Decision Support Systems 42: 4, pages 2164-2185.

Gollmann, D. (1999). *Computer Security*. Worldwide Series in Computer Science. publisher Wiley.

Griffith, R. & Kaiser, G. (2006). *A Runtime Adaptation Framework for Native C and Bytecode Applications*. In Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC '06). IEEE Computer Society, Washington, DC, USA, pages 93-104.

Kavulya, S. P., Joshi, K., Giandomenico, F.D. & Narasimhan, P. (2011). *Failure Diagnosis of Complex Systems*. In K. Wolter, A. Avritzer, M. Vieira & A. van Moorsel (Eds.), *Resilience Assessment and Evaluation of Computing Systems*, Springer Berlin Heidelberg.

Kephart, J. (2003). *An architectural blueprint for autonomic computing.* IBM White Paper.

Keromytis, A.D. (2007). *Characterizing self-healing software systems*. In Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS).

Nami, M.R. & Bertels, K.L.M. (2007). *A Survey of Autonomic Computing Systems*. 3rd International Conference on Autonomic and Autonomous Systems (ICAS 2007).

NIST (2013). Glossary of Key Information Security Terms. Revision 2.

O'Sullivan, P., Anand, K., Kotha, A., Smithson, M., Barua, R. & Keromytis, A.D. (2011). *Retrofitting Security in COTS Software with Binary Rewriting*, In J. Camenisch, S. Fischer-Hübner, Y. Murayama, A. Portmann & C. Rieder (Eds.), *Future Challenges in Security and Privacy for Academia and Industry*, Springer Berlin Heidelberg.

Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J. & Treuhaft N. (2002). *Recovery Oriented Computing (Roc): Motivation, Definition, Techniques*. Technical Report. University of California at Berkeley, Berkeley, CA, USA.

Perino, N. (2013). *A framework for self-healing software systems*. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, pages 1397-1400.

Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y, Ernst, M.D. & Rinard M. (2009). *Automatically patching errors in deployed software.* In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, pages 87-102

Portokalidis, G. & Keromytis, A.D. (2011). *REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points*. Proceedings of the 6th International Workshop on Security (IWSEC), pages 16-32, Tokyo, Japan.

Psaier, H. & Dustdar, S. (2011). *A survey on self-healing systems: approaches and systems.* Computing 91(1): pages 43-73.

Sidiroglou, S., Locasto, M.E., Boyd, S. W. & Keromytis, A.D. (2005a). *Building a reactive immune system for software services*. In Proceedings of the annual conference on USENIX Annual Technical

Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, pages 11-11.

Sidiroglou, S., Giovanidis, G. & Keromytis, A.D. (2005b). *A dynamic mechanism for recovering from buffer overflow attacks*. In Proceedings of the 8th international conference on Information Security (ISC'05). Springer-Verlag, Berlin, Heidelberg, pages 1-15.

Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J. & Keromytis A.D. (2009). *ASSURE: automatic software self-healing using rescue points*. SIGPLAN Not. 44,3, pages 37-48.

Stroupe, A. W., Singh, S., Simmons, R., Smith, T., Tompkins, P., Verma, V., Vitti-Lyons, R. & Wagner, M.D. (2002). *Technology for autonomous space systems*. Technical Raport CMU-RI-TR-00-02. The Robotics Institute, Carnegie Mellon University, Pittsburgh.

Swimmer, M. (2007). *Using the danger model of immune systems for distributed defense in modern data networks.* Computer Networks: The International Journal of Computer and Telecommunications Networking. Volume 51, issue 5, pages 1315-1333.

The History of Vaccines.(2013). *The Human Immune System and Infectious Disease*. The College of Physicians of Philadelphia, http://www.historyofvaccines.org/content/articles/human-immune-system-and-infectious-disease [2013-12-05].

Veríssimo, P. (2002). *Intrusion Tolerance: Concepts and Design Principles. A Tutorial*. Technical Raport. http://hdl.handle.net/10455/2988 [2013-12-03]

Westmark, V.R. (2004). *A definition for information system survivability.* Proceedings of the 37th Annual Hawaii International Conference on System Sciences.

Yuan, E., Esfahani, N. & Malek, S. (2010). *A Systematic Survey of Self-Protecting Software Systems.* ACM Transactions on Autonomous and Adaptive Systems.

Zavou, A., Portokalidis, G. & Keromytis. A.D. (2012). *Self-healing multitier architectures using cascading rescue points*. In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12). ACM, New York, NY, USA, pages 379-388.

FOI, Swedish Defence Research Agency, is a mainly assignment-funded agency under the Ministry of Defence. The core activities are research, method and technology development, as well as studies conducted in the interests of Swedish defence and the safety and security of society. The organisation employs approximately 1000 personnel of whom about 800 are scientists. This makes FOI Sweden's largest research institute. FOI gives its customers access to leading-edge expertise in a large number of fields such as security policy studies, defence and security related analyses, the assessment of various types of threat, systems for control and management of crises, protection against and management of hazardous substances, IT security and the potential offered by new sensors.

FOI
Swedish Defence Research Agency
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se