Software Engineering for Variability-Intensive Systems,CRC Press/Taylor & Francis Group (c) 2018

Towards Self-securing Software Systems: Variability Spectrum

Mohamed Abdelrazek[1], John Grundy[2] and Amani Ibrahim[1]
[1]School of IT, Deakin University, Geelong, Australia
[2]Faculty of IT, Monash University, Melbourne, Australia
mohamed.abdelrazek@deakin.edu.au; amani.ibrahim@deakin.edu.au;
john.grundy@monash.edu

Abstract

We describe a new variability-intensive system idea, the "self-securing software system". We describe how such a system works using a multi-tenant cloud application as a motivating example. This supports run-time composition, detects emergent attacks and vulnerabilities, and supports run-time updating to mitigate problems. We describe recent work we have done in architecting and proof-of-concept prototypes for aspects of such systems. We then describe current limitations and future work plans to address these.

1. Introduction

What can go wrong when a software system is under a cyberattack? The system becomes mal-functional; this could happen in many ways - *e.g.* the system might become completely unavailable due to service disruption, systems could be compromised, or confidential data could be exfiltrated. Businesses economic conditions amplify the relentless challenge of staying ahead of security vulnerabilities in their computing systems and infrastructure [9,11,17]. Hence, the need to build smart, adaptive systems that are secure. *Self-securing* (also called self-protecting or self-defending) is the ability of the software system to identify security holes/bugs or detecting current/potential attacks and automatically patch or stop to guarantee the availability, integrity and confidentiality aspects of the system and its data.

To enable self-securing systems, we need to: *(i)* package software security analysis and automated software security engineering in one ecosystem, and *(ii)* engineer systems to allow adaptability from the ground-up (architecture and design). Without adaptability, systems become hardwired and hard to respond to adaptation triggers including proactively self-secure [4,16]. Adaptiveness is achieved via engineering for variability points that allow for different system components - at different complexity levels from a class method to a system component - to be swapped in/out based on changes in the operational environment or customer needs and goals.

From a security point of view, variability is not always a good practice. From a security analysis perspective; it increases the potential of system vulnerabilities that might exist in these new modules, or in the connectors/interfaces between the modules [7,15]. From a security engineering perspective; variability is very useful, changes in user security requirements, risks, or new vulnerabilities detected can be mitigated/patched easily at either design-time (due to modularity nature of the system) as well as real-time adaptation [6,16].

In Software-as-a-Service (SaaS) applications – *e.g.* Salesforce.com – implementing a tenant-oriented security is a must. This is needed to enable the SaaS application to weave in security controls for different tenants depending on who is using the application/service – *e.g.* each tenant might need a different set of authentication/authorization controls. This adds a new

dimension of SaaS application variability, which is SaaS application security variability [13,14]. Other quality of service attributes can add more variability aspects to the system such as reliability-specific modules and performance critical modules that sacrifice resource efficiency for response time. This will eventually need to be managed through a SaaS variability management module.

In this short chapter, we present a proposal for the "*self-securing software system*" based on model-driven engineering using system-and-security description models for run-time configuration and weaving using an aspect-oriented programming and online security monitoring and analysis toolset. We also discuss how this model can be extended to SaaS applications where the same service/application need to satisfy all tenants' security requirements.

2. Motivating Scenario

Imagine a software system - Galactic ERP, developed by the SwinSoft company. GalacticERP is delivered as a SaaS application is hosted on a cloud platform called Blue Cloud and uses external services delivered on Blue Cloud and another cloud provider called Green Cloud. Galactic ERP is currently in use by three clients: Swin, Auck, and Super.

These three clients are operating in different domains, have different functional requirements, and have to comply to different security policies and requirements due to criticality of their business. Furthermore, imagine that the below threats have come true and vulnerabilities have been detected – e.g.:

- A root-kit attack on its infrastructure as a service e.g. Green Cloud
- Injection attacks, from tainted requests to services such as Get Currency or Build Workflow
- Poor isolation allowing compromise of one component to attack another e.g. between SwinSoft platform and GalacticERP
- Excessive privileges for one component allowing access to services of another e.g. between Green Cloud and Build Workflow

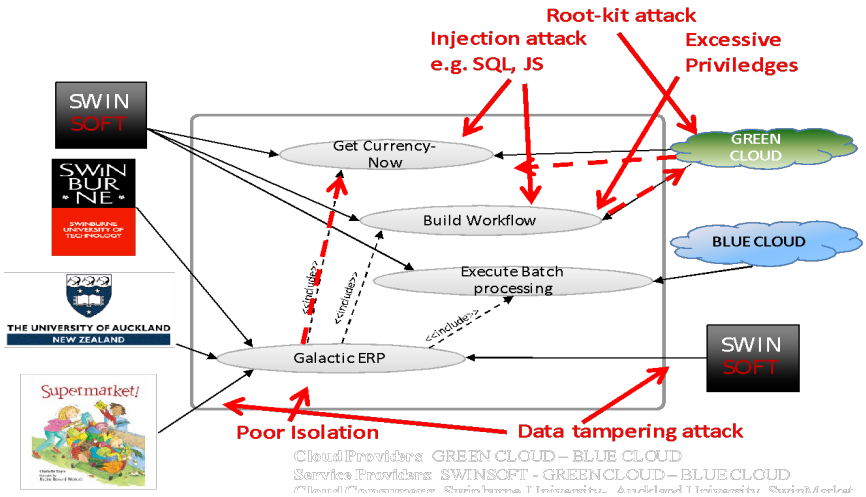

Figure 1. Example scenario of multi-tenant cloud application

Each system vulnerability or cyberattack needs to be mitigated promptly to protect the underlying system. Currently, most systems require this to happen at design time – through

major changes to the source code - with a patch and update of deployed services. This leaves systems vulnerable for a considerable period or requires them to be taken offline for fixing.

We have identified a set of key challenges in this domain:

- When we engineer cloud applications, we do not know what other applications and services will be deployed within the system, what hardware will be deployed on, what is the network underlying topology, etc.
- Stakeholder requirements change during deployment, especially for multi-tenant cloud apps, allowing for emergent requirements.
- New threats and vulnerabilities to attack are continually emerging and evolving.
- Design-time patching/re-deploying is a slow process that could leave a system vulnerable for a considerable period – even with the new DevOps practices, this still needs thorough testing.

We have identified a set of key requirements to address these challenges, which include:

- Identify emergent threats - even as its environment changes.
- Identify mitigations to the threats *i.e.* changes to the system configuration, code, deployment to mitigate the threat and protect it from attack.
- Self-adapt the application(s) using one (or more) mitigations while in use to counter the threat.

We describe several partial approaches to this solution in the following sections. We group these solutions as security analysis tools and capabilities including vulnerability analysis and security monitoring, and security engineering tools and capabilities including design time and runtime adaptations.

3. Static Vulnerability Detection

This is a part of our larger "*model-driven security engineering*" platform body of work [4]. The idea is to develop semi-formal definition (signatures) of OWSAP and CAPEC databases of security vulnerabilities using a domain specific language – currently, this is done in Object Constraint Language – OCL – enriched with constructs from the traditional software architecture/design/code concepts. Then we use these signatures to search for matches in code or in models associated with applications (*e.g.* configuration files and selection of application software). This approach can handle code vulnerability detection *e.g.* design and architecture vulnerability detection and several security "metrics" [2].

Figure 2 shows three different example code level vulnerabilities: Figure 2(a) code that is vulnerable to SQL injection; Figure 2(b) code that is vulnerable to an authentication by-pass; and Figure 2 (c) code that is vulnerable to improper authorization threat. Figure 3 shows example semi-formal signatures written as OCL constraints. These signatures describe, using a model of the target software system, how such vulnerabilities in Figure 2 can be found in the code base using static analysis.

We have developed a toolset that can take source code, a signature database, and analyse the source code for code locations vulnerable to the specified attacks in the semi-formal signatures. Figure 4 shows the basic analysis process. We take a target program (left) and a set of signatures (right) and then search for matching code or model signatures in the target. This search is informed by the platform characteristics (top). We produce a set of vulnerabilities and

their locations in the code base/model (bottom) as output. We have extended this approach to include design- and architecture-level model signatures, including configurations, architectural choices, and deployment platforms and associated application packages.

```
Public bool LogUser(string username, string password) {
    string query = "SELECT username FROM Users WHERE
    UserID ='" username " ' AND Password = '" + password + "'";
```

(a) Code vulnerable to SQL Injection attack.

```
if( Request.Cookies["Loggedin"] != true ) {
        if( !AuthenticateUser(Request.Params["username"],
                                Request.Params["password"] ) )
            throw new Exception("Invalid user");
}
DoAdministrativeTask();
```

(b) Code vulnerable to authentication bypass.

```
if( !AuthenticateUser( Request.Params["username"],
                       Request.Params["password"] ) )
        throw new Exception("Invalid user");
updateCustomerBalance(Request.QueryString["custID"], nBalance);
```

(c) Code vulnerable to improper authorisation.

Figure 2. Examples of common attacks.

Vul.	Vulnerability Signature (Simplified!!)
SQLI	Method.Contains(S : MethodCall | S.FnName = "ExecuteQuery" AND S.Arguments.Contains(X : IdentifierExpression | X.Contains(InputSource)))
XSS	Method.Contains(S : AssignmentStatement | S.RightPart.Contains(InputSource) AND S.LeftPart.Contains(OutputTarget))
Improper Authn.	Method.IsPublic == true AND Method.Contains(S : MethodCall | S.IsAuthenitcationFn == true AND S.Parent == IFElseStmt AND S.Parent.Condition.Contains(InputSource))
Improper Authz.	Method.IsPublic == true AND Method.Contains(S : Expression | S.Contains(X: InputSource | X.IsSanitized == False OR X.IsAuthorized == False)

Figure 3. Example formal vulnerability signatures.

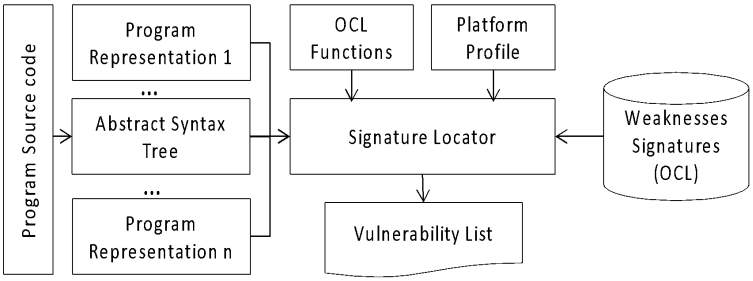

Figure 4. Vulnerability Analyzer.

4. Dynamic Application Monitoring

To detect system vulnerabilities and threats, we need to implement dynamic analysis solutions, as well. One dynamic analysis approach is designed to better capture run-time metrics and analyse these for evidence of incorrect application behaviour or usage from a security perspective [3]. In an equivalent way to the static analysis above, we formalise a number of metrics and security constraints of interest – similar to vulnerability signatures - using OCL. Figure 5 illustrates some examples of such security-related metrics of interest [10,1]. Some of these metrics are architecture and design-related metrics. Others, are runtime metrics – e.g. how many requests have been authenticated, how many of these passed (valid) authentication check, mean time between failed authenticated request, etc. We chose these as examples of the kinds of metrics a security engineer would like to monitor in order to proactively judge the current security status of multi-tenant cloud applications[1,12].

We then process a deployed application's architecture and deployment model and code base to determine where to add "probes" to gather the required run-time monitoring data. Figure 6 provides an outline of this process. A set of services are "wrapped" to provide access to their run-time behaviour via a set of probes (top). A set of metrics specifications determine where these probes need to be injected into the application at rum-time to monitor its behaviour (left). A probe generator takes the specifications, application model and configures and/or generates the probes (middle). The framework then captures the required run-time data, determines

exceptions based on the metrics specifications, and then determines possible attack and vulnerability mitigations that can be actioned and reports these (right).

Metric	Signature
Information Disclosure	context Method inv InfoDisclosure: Let access : Request := self.Requests->last() in Let authorized : Response := self.AuthorizationControl.Responses-> select(R| R.IsValid = True AND access.UserID = R.UserID)->last() in IF (authorized) THEN true ENDIF
Chinese Wall	Let Subject := Classes->select(Name = 'Subj')->first() in Let Obj: Class := Classes->select(Name = 'Object')->first() Let mthdCall : Request := self.Requests->last() in Let mthdReturn: Response := self.Responses->last() in Let access : Request := self.Requests->last() in IF (access.RequestTime > mthdCall.RequestTime and access.RequestTime < mthdReturn.ResponseTime) THEN Not self.Conflictlist->exists(R| R = access.Target)
Restrict System Calls	Let SystemCalls : Request := Classes->select(Name = 'SystemHandler')->first().Requests()->last() in IF (SystemCalls <> null) THEN false ENDIF
Separation of Duties	Let xReq : Request:= Requests(Entity = 'MthdX') in Let yReq : Request:= >Requests(Entity = 'MthdY') in Let zReq : Request:= >Requests(Entity = 'MthdZ') in IF (xReq.UserID = yReq.UserID and xReq.Target = yReq.Target Or xReq.UserID = zReq.UserID and zReq.Target = zReq.Target Or yReq.UserID = zReq.UserID and xReq.Target = yReq.Target) THEN false ENDIF
Authenticated Requests	context System inv AuthenticatedRequests: self.AuthenticationControl.Requests->select()->count()/ self.Request->select()->count()
Authentic Requests	context System inv AuthenticRequests: self.AuthenticationControl.Response->select(R | R.IsValid = true)->count()/ self.AuthenticationControl.Request->select()->count()
Last(10) Authz. Reqs	context System inv Last10AuthzCtl: self.AuthorizationControl.Requests->select()->Last(10)
Top(10) admin Requests	context System inv Top10AuthnCtl: self.AuthenticationControl.Responses->select(R | R.UserID = 'Admin')->count()
Mean Time Between Unauthentic Request	context System inv MTBUnauthenticRequests: self.AuthenticationControl.Responses->select(R | R.IsValid = false)->differences('Measurementtime')-> sum() / self.AuthenticationControl.Responses->select(R | R.IsValid = false))->count()
Authenticated Requests Trend	context System inv Authenticated RequestsTrend: self.AuthenticatedRequests.Differences('AuthenticatedRequests')->sum() / self.AuthenticatedRequests-> count()
MTBUR Over Systems	context System inv MTBUROverSystems: self.MTBUnauthenticRequests->sum()/ self.MTBUnauthenticRequests->count()

Figure 5. Examples of security monitoring metrics formal signatures.

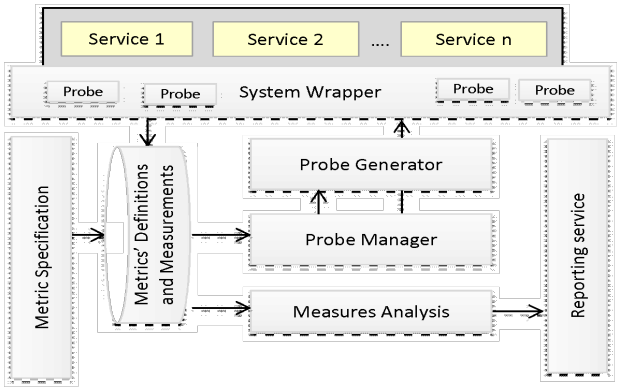

Figure 6. Run-time probe generation process.

5. Run-time Vulnerability Mitigation

Finally, we must secure our vulnerable or under threat applications. Once we have found a vulnerability using one of the above techniques, we need to determine how to fix, mitigate or raise an alarm. We identify feasible modifications to the target application to address the vulnerability: these might be to inject code to fix an SQLI vulnerability *e.g.* input sanitisation code; upgrade application software deployed with the services to ensure the latest version is being used with it; reconfigure service end points to apply more secure encryption, key management, auditing or other security services; make use of new, improved APIs, libraries and/or third party security solutions. Figure 7 shows examples of code updates to mitigate some of the code-level vulnerabilities illustrated earlier. In this example, we use an aspect-oriented

toolset, “re-aspects”, which describe code updates to make (add, remove, modify) using their own formal OCL signatures [3].

We then update the application to address the discovered vulnerability, security flaws or counter attack scenario. This may include making run-time code changes, making run-time configuration changes, redirecting requests to different third-party security solutions, or deploying different third-party security solutions. It may also in some cases mean restricting access to parts of the system that remain vulnerable, raising alarms, or worst-case scenario, taking a system or some services off-lime. We validate that the originally identified vulnerability has been addressed using an appropriate testing framework e.g. by running attack scenarios on the vulnerable service, re-running vulnerability analysis on the updated code base, collecting new monitoring metrics etc.

Figure 8 (a) shows an example of the toolset we have developed to support these re-aspect run-time code update and Figure 8 (b) an example of the tool in use. (1) We first convert a target code base into Abstract Syntax Tree format (AST). (2,3) Formal signatures are constructed using a modelling tool, an example shown in the screen dump in Figure 8(b). (4) We then use this set of re-aspect formal signatures to locate parts of the code base to modify. (5) We then determine the full impact analysis of making the code change, which may be at statement, method, class or caller (system-wide) levels, to determine the full set of code changes needed. (6) Finally we propagate changes to the code of the application. We have a .NET based tool that is able to do this at run-time, modifying compiled CLI code of the application [3].

```
if( Request.Cookies["Loggedin"] != true ) {
    if( !AuthenticateUser(Request.Params["username"],
                          Request.Params["password"] ) );
        throw new Exception("Invalid user");
}
DoAdministration();
```

(a) Replacing authentication bypass vulnerable code

```
if( !AuthenticateUser( Request.Params["username"],
                       Request.Params["password"] ) )
    throw new Exception("Invalid user");
if( !AuthorizeUser( Thread.CurrentPrincipal,
          (new StakeFrame()).GetMethod().Name,
          (new StakeFrame()).GetMethod().GetParameters() ) )
    throw new Exception("User is not auhorized");
updateCustomerBalance(Request.QueryString["cID"], nBalance);
```

(b) injecting code to fix improper authorization

```
Inputsanitizer( (new StakeFrame()).GetMethod().GetParameters() );
string query = "SELECT *  FROM USERS WHERE UserID = '"
+ EncodeForSQL(username) + "' AND password = '"
+ EncodeForSQL(password) + "'";
```

(c) Adding functions to sanitize inputs for SQLI

```
bool updateCustomerBalance(string custID, decimal nBalance) {
    if(!AuthenitcateUser( username, password)) return false;
    if(!AuthorzUser(username, "updateCustBalance")) return false;
    LogTrx(username, dateTime.Now, "updateCustomerBalance");
    Customer customer = Customers.getCustomerByID(custID);
    customer.Balance = nBalance;
    Customers.SaveChanges();
    LogTrx(username, dateTime.Now, "updateCustBalance done");
}
```

(d) deleting obsolete security code.

Figure 7. Code updates to mitigate vulnerabilities.

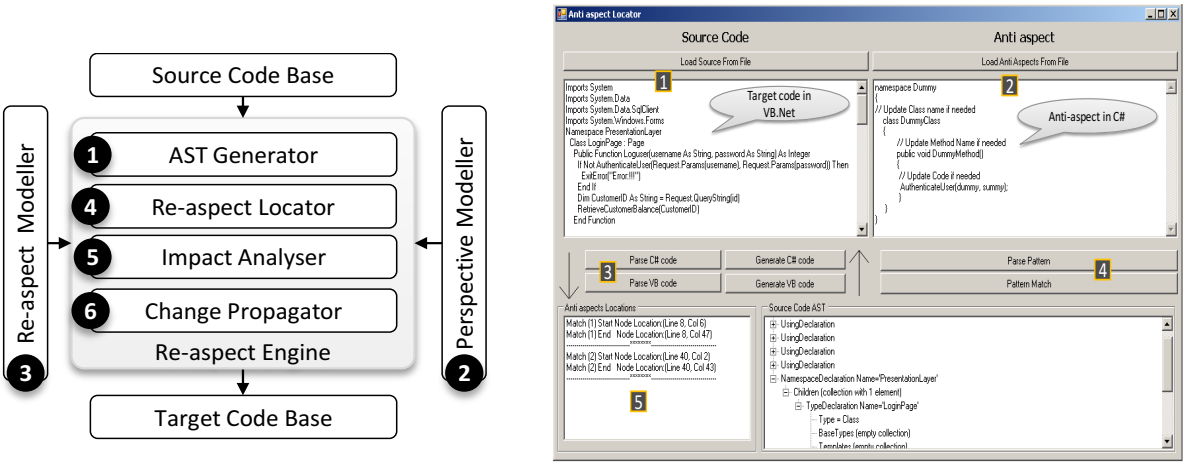

Figure 8. (a) code updating process; (b) SMART code analysis and updating tool.

The other way to update an application is to use its built-in variability support features. Figure 9 shows our run-time reconfiguration framework, MDSE@R (Model-Driven Security Engineering @ Run-time), being used to modify configurations of a deployed system to mitigate discovered security flaws. (1) Vulnerabilities are discovered using static or dynamic analysis; (2) Mitigation actions are identified to correct the discovered problem; (3, 4) a set of updates to our deployed application configuration are planned; (3, 5) a set of updates to our deployed application security handling configurations are planned; (6, 7) these application and/or application security configuration updates are made; and (8) the reconfigured application makes use of appropriate security services as it is used.

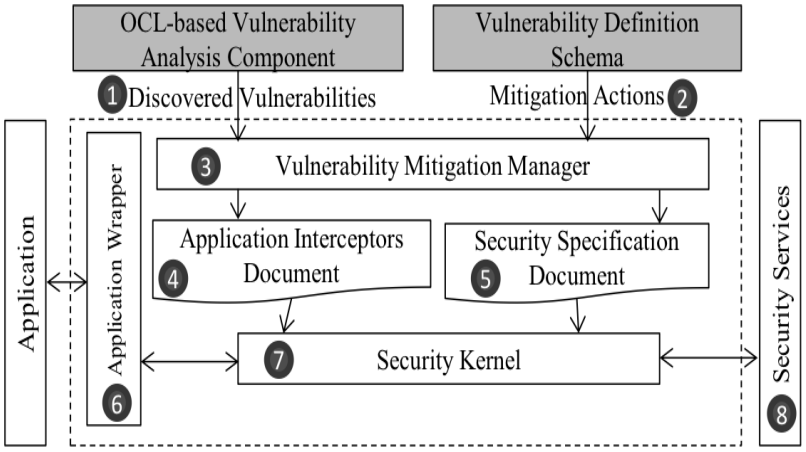

Figure 9. Run-time configuration updates to mitigate vulnerabilities.

6. Outstanding Challenges

To date we have built several proof-of-concept tools that implement the above aspects of variability intensive self-securing software systems. We have found some interesting challenges we trying to adopt this approach. Firstly, run-time updating of applications is inherently challenging. Running code/configured software applications must be modified with care to ensure no catastrophic failure during update but also to handle partially completed processes in sensible ways. Even design time modifications are very challenging when it comes

to statement level code modifications. Changes need to be thoroughly tested for both functionality as well as security defects.

Security analysis is a very complex task. It covers threat analysis, attack analysis and vulnerability analysis. Many techniques have been developed to detect vulnerabilities using both static and/or dynamic analysis. The rule-based vulnerability analysis techniques are effective but not enough given that every vulnerability has many ways to be present and many ways to be detected. There is a trend to start using machine learning techniques to try and detect security defects by learning what is best security practice and what is not.

Runtime security monitoring is always a necessary security capability recommended by most of the security standards. Although, there is always a process to follow to develop security metrics, but there is no recipe of a list of metrics to use, when to use these metrics, what to do when a metric is not within norm or what would be a norm. This gets even complicated in a SaaS application where we have multiple tenants, each have their own security requirements and measures.

Comparing systems within the same domain across different security dimensions can be both deceiving and inaccurate. Vulnerability counts vs security metrics vs actual meaning for the vulnerability of the application varies greatly and the impact of the vulnerability on the application and its users can vary greatly. For example, simple read of a highly confidential but improperly protected data element may be far more damaging than complex exploits or severe denial-of-service attacks. The scale of the system can also have major issues on the viability of some of our techniques where they are computationally expensive to run at run-time. This means some analyses are not real-time but can only detect vulnerabilities sometime after the fact. A minor change to the configuration of the application or its deployment environment *e.g.* a new service or tenant, can have a major impact on the outcome of the security analysis. For example, a new tenant for the exact same multi-tenant cloud application may wish to make use of their own preferred security service, which introduces a whole new potential vulnerabilities or attack surfaces.

We have found run-time dynamic security analysis that are still an emergent area with it being still unclear for many potential vulnerabilities what the “right” metrics are to capture, analyse and determine threat. Security is a “cost” in that it introduces additional performance overhead and complexity to the application. Run-time adaptation to mitigate detected threats introduces further overheads and complexities into the system.

Finally, to implement run-time mitigation of emergent threats requires architectures, designs, code and deployments that support this in highly flexible ways and yet maintain high application performance and function. A highly variable system at run-time introduces its own risks in that compromise of the variability supporting architecture components are their own major security vulnerabilities.

We are working on several areas to help us realise the concept of the self-securing software system. We are carrying out further formalisation of the OWSAP and CAPEC databases of security vulnerabilities to enable us to detect further security flaws and vulnerabilities in target applications. Similarly, we are working on further mitigations for these including formalised models of the mitigations that can be automatically actioned at run-time.

We are applying deep learning to static and dynamic vulnerability detection vs our current rule-based (DIGGER, SMART) approaches and are using statistical-based log analysis approaches to compliment these. These will allow us to train vulnerability detection models from examples and, we hope, to also allow us to update these trained models as new threats emerge. These approaches imply we have good training sets and vector-based models for our applications, both areas of our current work.

We believe that better supporting tenants to specify their security requirements is both essential for multi-tenant cloud applications, but very challenging [5]. We are working on improved tenant security modelling tools that will be used by the self-securing application to determine the actual security requirements to enforce. Finally, zero-day threat detection at the IaaS level extremely hard but we are working on how to apply this approach to IoT security analysis and mitigation. This is needed to self-secure new IoT-based systems.

7. Threats to Validity

To date we have built prototypes of our three approaches to static analysis, dynamic monitoring and runtime adaptation frameworks [1,2,3]. We have evaluated these using a set of open source cloud applications written in .NET. A major limitation of the static checking approach is its use of rule-based signatures, requiring expert authoring, updating to new platforms and configurations, and fragility when applied to new code patterns and programming languages. We are exploring use of deep learning-based training of the vulnerability analyser where we train the learner on the vectorised code base, enabling us to retrain it on new examples and avoiding expert rule authoring [8]. Similarly our current dynamic monitoring approach may benefit from an approach of training a recongizer on examples of trace data rather than a rule based approach. Another limitation of the dynamic monitor is generating suitable probes and integrating probes into the runtime environment of existing cloud applications. Many are not architected to enable this level of monitoring nor runtime update of monitors. Overheads of monitoring are a classic problem of such approaches which we also need to address. Finally, our re-aspects framework has proven well suited to runtime adaptation of .NET platform multi-tenant cloud applications but such runtime update is fraught with challenges. Not least is handling update during processing of data by the target application and efficient update is a further challenge. We are exploring use of micro-service based architectures to allow more precise runtime modification of services and also because such architectures are designed for such small scale, runtime modifications.

8. Summary

We have described a concept of the "self-securing software system". This is a variability intensive system that *(i)* runs static and dynamic analysis techniques to determine if a running multi-tenant cloud application is vulnerable to attack; *(ii)* identifies potential run-time mitigations, or fix-ups to the application to counter the vulnerability; and *(iii)* carries out run-time code and/or configuration updates of the application to force these mitigations.

Acknowledgements

Support for this research from Swinburne University of Technology, Deakin University, Monash University and ARC Discovery projects DP170101932 and DP140102185 is gratefully acknowledged by the authors.

References

1. Abdelrazek, Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. "Improving Tenants' Trust in SaaS Applications Using Dynamic Security Monitors." *Engineering of Complex Computer Systems (ICECCS), 2015 20th International Conference on*. IEEE, 2015.
2. Almorsy, Mohamed, John Grundy, and Amani S. Ibrahim. "Automated software architecture security risk analysis using formalized signatures." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.
3. Almorsy, Mohamed, John Grundy, and Amani S. Ibrahim. "Supporting automated software re-engineering using re-aspects." *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012.
4. Almorsy, Mohamed, John Grundy, and Amani S. Ibrahim. "Adaptable, model-driven security engineering for SaaS cloud-based applications." *Automated software engineering* 21.2 (2014): 187-224.
5. Almorsy, Mohemed, John Grundy, and Amani S. Ibrahim. "Collaboration-based cloud computing security management framework." *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011.
6. Altekar, Gautam, et al. "OPUS: Online Patches and Updates for Security." *USENIX Security Symposium*. 2005.
7. Cetina, Carlos, et al. "Autonomic computing through reuse of variability models at runtime: The case of smart homes." *Computer* 42.10 (2009).
8. Dam, Hoa Khanh, et al. "DeepSoft: A vision for a deep model of software." *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016.
9. Devanbu, Premkumar T., and Stuart Stubblebine. "Software engineering for security: a roadmap." *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000.
10. Hayden, Lance. *IT Security Metrics: A Practical Framework for Measuring Security & Protecting Data*. McGraw-Hill Education Group, 2010.
11. Krutz, Ronald L., and Russell Dean Vines. *Cloud security: A comprehensive guide to secure cloud computing*. Wiley Publishing, 2010.
12. Luna, Jesus, et al. "A security metrics framework for the cloud." *2011 Proceedings of the International Conference on Security and Cryptography (SECRYPT)*. IEEE, 2011.
13. Mellado, Daniel, Eduardo Fernandez-Medina, and Mario Piattini. "Security requirements variability for software product lines." *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. IEEE, 2008.
14. Myllärniemi, Varvana, Mikko Raatikainen, and Tomi Männistö. "KumbangSec: An Approach for Modelling Functional and Security Variability in Software Architectures." *VaMoS*. 2007.
15. Olaechea, Rafael, et al. "Modelling and multi-objective optimization of quality attributes in variability-rich software." *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*. ACM, 2012.
16. Petkac, Mike, and Lee Badger. "Security agility in response to intrusion detection." *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000.
17. Zissis, Dimitrios, and Dimitrios Lekkas. "Addressing cloud computing security issues." *Future Generation computer systems* 28.3 (2012): 583-592.