

*Computer Science
Technical Report*



Analysing Requirements to Detect Latent Security Vulnerabilities

Curtis C.R. Busby-Earle
University of the West Indies
curtis.busbyearle@uwimona.edu.jm

Robert B. France
Colorado State University
france@cs.colostate.edu

November 11, 2013

Colorado State University Technical Report CS-13-108

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: <http://www.cs.colostate.edu>

Analysing Requirements to Detect Latent Security Vulnerabilities

Curtis C.R. Busby-Earle

University of the West Indies
Mona, Jamaica

Email: curtis.busbyearle@uwimona.edu.jm

Robert B. France

Colorado State University
Fort Collins, USA

Email: france@cs.colostate.edu

Abstract—To fully embrace the challenge of securing software, security concerns must be considered at the earliest stages of software development. Studies have shown that this reduces the time, cost and effort required to integrate security features into software during development. In this paper we describe a loop-hole analysis technique for uncovering potential vulnerabilities in software requirements specifications and describe its use using an example.

I. INTRODUCTION

Security requirements are seldom explicitly stated at the outset of a project [1]. Typical security considerations at the requirements development stage are: confidentiality of sensitive information [2]; potential system threats and exploits [1], [3], [4]; privacy and trust concerns [5], [6]; and profiles of potential attackers [4], [7]. Requirements engineers and security experts attempt to address security issues by using approaches such as misuse cases [1], abuse cases [4], UMLSec [7], the SQUARE method [8], KAOS [9] and security patterns [10]. Many of the current approaches, however, rely heavily on the expertise and subjective judgement of security professionals. As an example of the subjectivity that comes into play when using these approaches, consider the use of misuse cases. A misuse case is a description of behavior that should not occur in a system. Misuse cases are described alongside use cases. The development and analysis of misuse cases may proceed as follows:

- 1) Describe the services that the users want, regardless of any security considerations. Use cases are used for this purpose
- 2) Introduce the major misuse cases and mis-actors. A misuse case is a special kind of use case, initiated by a mis-actor, that describes behavior that the system/entity owner does not want to occur.
- 3) Investigate the potential relations between misuse cases and use cases, and describe as use-case includes-relations. This step is quite important since many threats to a system can be achieved by using the system's normal functionality.
- 4) Introduce new use cases that detect or prevent misuse cases.

Steps two and three are key to the process, but they are subjective and the results of their application are completely

dependent upon a security expert's judgement. We are aware that there are many flaws that currently require human expertise to uncover, but there are, nevertheless, security flaws that can be identified using less subjective methods during the early stages of software development [11]. Examples of such flaws are SQL and command injection which take advantage of improper input validation. Input validation functions can be specified as early as the requirements phase.

Although a requirements engineer has the task of specifying what an intended software system should do, a requirements engineer is not expected to be a security expert. Security is usually grouped with and considered as, a non-functional requirement [9], [12]. Non-functional requirements have traditionally been included during the coding, implementation or maintenance stages of software development i.e. at the end of the development process [13].

In this paper we present a technique that provides a means by which the knowledge of security experts can be captured, retained, used, refined and shared among requirements engineers and other software engineering practitioners, thus assisting them in their task of considering security concerns at the earliest stages of software development. The expertise is captured in the form of a type of dependency between requirements. The technique is incorporated into a less subjective approach we developed to uncover potential vulnerabilities that can augment those that require human expertise. The approach is summarised as follows:

Given a set of use cases, each scenario step is transformed into a concise, normalised statement that captures interactions in a structured analysable form. The next step involves using pre-defined dependencies between elements of normalised statements to identify potentially vulnerable interactions across the normalised statements. This is done by forming the transitive closure of the dependency relationship and highlighting the interactions involving the related elements that violate security or other policies.

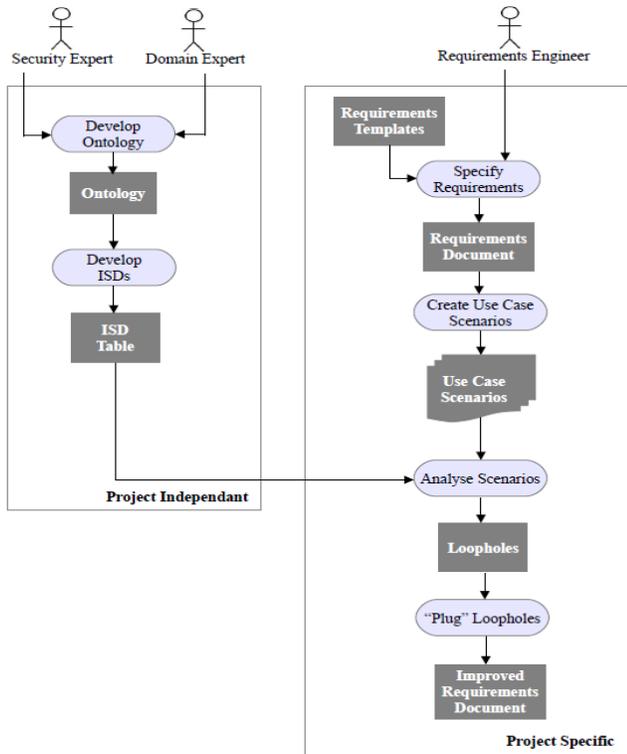


Fig. 1. Overview of the Loop-hole Analysis

II. BACKGROUND

The proposed technique, called the loop-hole analysis, addresses the problem of functional fixation in the context of requirements analysis. Functional fixation is the inability to see uses for something beyond its presented use [14]. In other words, it is the belief that something can only be used for its stated purpose. In the context of software systems we can and do use functionality provided by software in unintended ways.

For example, a Windows XP user with no administrator privileges can acquire them by creating a shortcut to IE6, enable the 'Run with different credentials' option, and open a shell as a local administrator. From a security standpoint, each step involved in accomplishing the task is allowed, but this particular usage leads to undesirable situations. Another example is provided by Linux, Android and other UNIX-based operating systems where automatic file completion is a very useful feature but, it also helps intruders to find target files more quickly [1]. Everyday objects, including software, may fulfill their stated goals and yet, may allow undesirable behaviour.

The loop-hole analysis seeks to identify such undesirable interactions during the process of specifying requirements (see Figure 1). Uncovering and mitigating potential vulnerabilities during requirements analysis, reduces the time, effort and cost of fixing these problems, when compared to addressing them later in the development process [7], [8]. At such an

early stage of development, it is the potentially vulnerable interactions in use case scenarios that must be identified. Use case scenarios typically describe a system from a user's perspective, and thus focus on user-system interactions. A scenario is an ordered set of interactions between partners, usually between a system and a set of actors external to the system.

Scenarios are typically written in a natural language to facilitate their understanding by as many of the stakeholders of an intended system as is possible. Such general understanding by many is furthered by consistency, precision and the lack of ambiguity in the expression of requirements. Consistency means that requirements should not have contradictory definitions [15]. Imprecision in a requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its interpretation [15]. For example, the requirement shown here seems to conform to good requirements specification practice,

The user shall be able to create a password

But what should it mean to an engineer, in the context of security, to "create"? With respect to passwords and other types of authentication data, the term "create" may mean different things to different people. To one engineer, the creation of a password simply involves its conceptualisation and subsequent entry by a user. To another, it may mean the same but be subject to restrictions in length and composition, and understood to be one factor in a multi-factored approach to authentication.

We believe that security considerations in software development have necessitated the specificity in meaning of terms when developing requirements, and in particular for the process of identifying potential vulnerabilities in software systems during requirements engineering. We suggest that such specificity can be achieved using a consistent description of requirements terms. The number and variety of terms is however, potentially large due to synonymy. For example, edit, change and modify all have similar meanings. Therefore, rather than attempt to describe every term, we suggest a model that will specify the terms used by a requirements engineer that incorporates security considerations.

The primary issue, with incorporating aspects related to the security of a system during the requirements engineering process is that they typically remain unstated or unrecorded i.e they are included implicitly [16]. A system's security requirements therefore tend to remain excluded from requirements specifications or are the subject of many an assumption by members of development teams. What engineers and developers alike assume, in part due to implicit requirements, can (and often does) lead to varied

interpretations of explicit requirements. Implicit requirements should be made explicit.

The technique we describe aims to improve a requirements document with respect to its security requirements by attempting to uncover implicit security requirements by a process that uses an *imposed security dependence* (ISD) [17]. We impose security dependencies between requirements terms by stating that the inclusion of one term in a requirement necessitates the inclusion of another term and requirement, upon which a dependency has been defined based on security considerations. For any two requirements terms (RT) α and β , we define an ISD as,

RT α has an ISD on RT β when the use of α in a requirement dictates the use of β in a separate requirement.

Two requirements therefore have an ISD relation when an ISD exists between requirements terms contained in each. We represent an ISD between two RTs using the $-]$ symbol and parentheses. $\neg]$ indicates that two RTs do not have an ISD relation.

Commonly used requirements terms and their imposed security dependencies are created by a requirements engineer/domain expert and a security expert (see Figure 1). These experts use their knowledge and experience to create an ontology of requirements terms that is independent of any specific project. The ontology is however, created within the context of a specific domain. The ontology is then used to develop a table of ISDs. The domain specific ISD table is stored for subsequent use in individual, domain specific projects. Once such a table of terms and ISDs has been created, the knowledge and expertise of the domain and security experts will be captured and retained in the ISD table. Through its use in various software development projects, the table and its content can be shared, refined and improved upon by other practitioners.

III. THE LOOPHOLE ANALYSIS

To uncover the types of interactions previously discussed we use the notion of a path. A path is a sequence of use case scenario steps, where the sequence can cut across many use cases. Use case scenarios typically describe a system from a user's perspective, and thus focus on user-system interactions. A scenario is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. We define path fixation as the belief that the simple paths described in a specification document are the only ones that will exist in the implemented system. The discovery of paths that overturn such beliefs is the basis of our loophole analysis. Piessens defines a vulnerability as any aspect of a computer system that allows for breaches in its security policy [18]. In the context of requirements analysis, loopholes are

paths that lead to violations in security or other policies that govern system behaviour.

A. Overview of the Technique

The input to our technique is a requirements document consisting of a set of requirements statements. The steps involved in the loophole analysis are given below:

- 1) Develop use case scenario descriptions from the requirements statements and then transform each use case step into a more concise, normalised form that focuses on the interactions embodied in the requirements statement.
- 2) Analyse the scenarios, using dependency relationships between elements of the normalised statements, to identify implicit security requirements. These security requirements are then explicitly stated.
- 3) Analyse the scenario descriptions with security requirements for loopholes.
- 4) Improve the requirements document by including requirements that rectify or "plug" any discovered loopholes.

We developed a prototype tool called the Secure Requirements Writer (SECRET) to assist engineers in completing these activities. In the following sections we describe these steps in greater detail and discuss the SECRET.

B. Converting Requirements Statements

Given the input set of requirements statements, a requirements developer uses templates to create use case descriptions that describe interactions captured by the initial requirements statements. Each use case and its constituent steps are associated with reference numbers that indicate the ordering of steps within a use case. Each use case step is then converted into a concise form that presents the interactions it describes in a structured, analysable form. The result is called an access control policy (ACP).

Using the prototype tool we developed, a requirements engineer creates use case scenarios by completing and grouping parameterised statements. These statements in turn, are created by choosing an appropriate template from the tool's collection and filling-in the chosen template by selecting arguments from drop-down lists. The templates are created by domain experts. For example, the template

The <subject> shall be able to <action>
<object> within <quantity> <time unit(s)>
of <event>

can be completed and used to create the following use case statement/requirement,

The driver shall be able to stop the automobile within 3 seconds of applying the brakes

The arguments used to complete the template are the requirement terms upon which security dependencies are imposed. They are stored in the tool's ISD table. The selected arguments are then copied and the tool generates the associated ACP. We call these arguments ACP arguments.

An ACP describes the action a subject performs on an object as either a capability or a constraint [19]. The ACP format is based on the four important characteristics of a good notation that are described by Iverson [20]. The characteristics of such notations are that they should,

- 1) allow for convenient expression of interactions
- 2) facilitate some degree of reasoning through subordination of the details inherent in requirements specifications
- 3) facilitate suggestivity of relationships among expressions
- 4) be economical - a relatively small vocabulary with simple grammatical rules

In general, the format of an ACP is,

$$ref[subject \ clarifier : action \ clarifier : object \ clarifier]_{cap/con} | comment$$

where:

- *ref* is a reference to a use case from the specification document from which the ACP was created
- *subject* is a type of actor that is capable of using an object. Types of actors can be users, processes or systems
- *action* describes a task performed on an object
- *object* is a system or application resource (something that can be used)
- *clarifier* identifies an attribute or instance of a subject, action or object and is used for refinement or distinction
- *cap/con* identifies whether the requirement is an intended system capability or constraint
- *comment* is a single line elucidation or exemplification of the requirement

For example, the use case statement,

2.1.5 The CFO shall have write access to the inventory file,

would be expressed in the ACP format as,

$$2.1.5[user_{CFO} : access_{write} : file_{inventory}]_{cap} |$$

where the subject is 'user_{CFO}', the action is 'access_{write}' and the object is 'file_{inventory}'. In the ACP format, CFO, write and inventory are clarifiers. The clarifiers and other variables (subject, action, object, capability or constraint, and the requirement reference) are the parameters of an ACP [19]. By transforming a requirements document to a set of ACPs

we are able to reduce its size while capturing the information related to interactions that can be used to identify potential vulnerabilities.

Template arguments are stored in an ISD table. Table 1 depicts a subset of typical entries in an ISD table. The 'Type' attribute corresponds to one of the ACP format's parameters subject, object, action or clarifier, while 'Bound to' indicates another ACP argument that the one being defined must be used with. Only clarifiers can be bound to other ACP arguments.

TABLE I
SAMPLE ISD TABLE ENTRIES

TERM	TYPE	DESCRIPTION	CLARIFIER REQ'D	BOUND TO	ISD
access	action	a right provided to an actor by the intended system	yes	-	-
verify	action	a process of comparison of a ACP argument or combination of ACP arguments with an associated standard of the intended system	no	-	-
write	clarifier	a process of recording data into an object by an actor	no	access	verify

Returning to example requirement statement 2.1.5, and referring to Table 1, ACP argument 'write' has an imposed security dependence on another argument, 'verify'. This dependency causes SECRES to generate a requirement stating that the system must verify the CFO has the permission to write to the inventory file. The ACP that would be generated would be,

$$2.1.5.1[system : verify : file_{inventory}]_{cap} | CFO \ write$$

ISD relations are neither reflexive nor symmetrical but are transitive i.e.,

$$(ACP1 \rightarrow ACP1), \quad (1)$$

$$(ACP1 \rightarrow ACP2) \rightarrow (ACP2 \rightarrow ACP1), \quad (2)$$

$$(ACP1 \rightarrow ACP2) \text{ and } (ACP2 \rightarrow ACP3) \Rightarrow (ACP1 \rightarrow ACP3) \quad (3)$$

C. The Loophole Algorithm

Having converted a specification document into a more concise form and developed a more complete set of requirements, we analyse them for loopholes by examining ACPs and their successors. An ACP's successors are those requirements that are directly reachable from it, i.e. they are elements of the sequence of activities described in use case scenarios that form a path.

We examine a set of requirements by defining a relation R . Let D be the set of all ACPs that are obtained from a particular requirements document, and R be a relation that maps an element of D to its successor element(s).

To represent the possible transitions from one requirement to another we defined R with the following properties,

$$R : D \times D \quad (4)$$

$$\forall r : D \bullet (r, r) \notin R \quad (5)$$

$$\forall r, q : D \bullet (r, q) \in R \not\Rightarrow (q, r) \in R \quad (6)$$

$$\forall r, q, s : D \bullet (r, q) \in R \wedge (q, s) \in R \Rightarrow (r, s) \in R \quad (7)$$

Because we are representing intended interactions in a system under development, intuitively, R is anti-reflexive, transitive (statements 5 and 7) and must therefore also be anti-symmetric (statement 6). Using R , a hierarchy of ACPs and their successors is created. These hierarchies can be conceptualised and depicted as simple digraphs. The requirement references (node values) and directed edges are used to represent the associations and sequence of interactions among requirements respectively.

We want to identify security policy breaches by utilising R . A policy is described by the allowed interactions of the intended system's users and objects i.e. the members of D . To identify breaches, we analyse representations of the activities by analysing R .

For this purpose, however, R is not sufficient as some of the possible paths are typically not explicitly included in a requirements document. These are the paths we are interested in. We therefore include the set of all reachable paths by finding the transitive closure of R .

The steps of the loophole analysis are as follows:

- 1) Represent the relation R as a binary matrix M .
- 2) Find the transitive closure of R , using the Floyd-Warshall algorithm [21]. Call this new relation R^* .
- 3) Represent R^* as a binary matrix M' .
- 4) Perform the bit-wise XOR of corresponding elements of M and M' . This will identify maplets created as a result of step 2 i.e. the indirect relationships that are not included in the original.
- 5) Where a 1 exists in M' but not M , excluding any that are reflexive or symmetric (statements 5 and 6), create a temporary ACP by combining the subject of the initial ACP of the path (start point) with the action and object of the final ACP (end point) of the path. In a two dimensional matrix representation of M' the start point will correspond to a row identifier and the end point a column identifier.
- 6) Compare each temporary ACP with every ACP in D expressed as a constraint.
- 7) A loophole (i.e. a vulnerability) exists when there is a match.

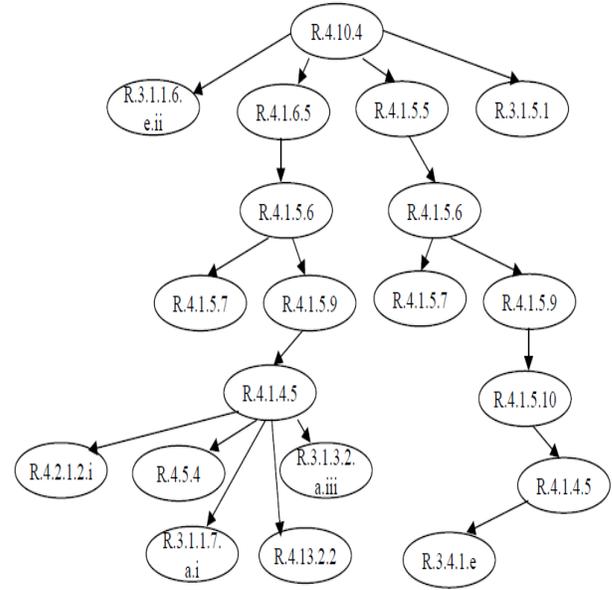


Fig. 2. A set of paths

D. Applying the Algorithm

Figure 2 is the digraph of a selection of paths from a requirements specification that describes required interactions between a user and a system when the user wants to login to perform services provided by the system. On one of these paths for example, the user logs in to the system, has her credentials checked, if accepted the system checks whether she has another open session and if there is no open session on the system, retrieves her permissions and displays her interface. Below is the set of associated ACPs that represent these activities.

$R.4.1.5.5[user : login : system]_{cap} |$
 $R.4.1.5.6[system : validate login : system]_{cap} |$
 $R.4.1.5.9[system : check : sessions active]_{cap} |$
 $R.4.1.5.10[system : retrieve : permissions user]_{cap} |$
 $R.4.1.4.5[system : display : interface]_{cap} |$

One of the options the ordinary user now has available is the activity called 'become administrator' (this is similar to the 'Run as Administrator' option available to ordinary users on some Windows operating systems, without requiring the presentation of credentials).

$R.3.4.1.e[user : select : process_{become_admin}]_{cap} |$

Selecting this option takes her along a path originating at R.4.1.6.5 (see Figure 2). This allows her to perform tasks such as the installation of small applications. All of her activities described thus far are completely legitimate and would comply with what is intended for the system under

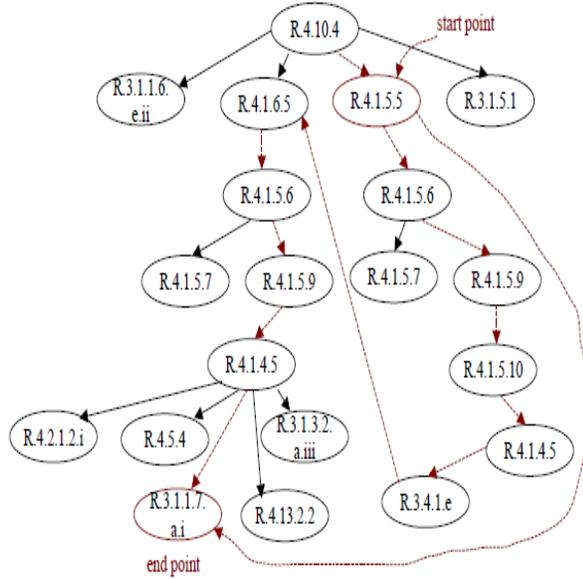


Fig. 3. The digraph of a loophole

development. When the loophole algorithm is now applied to the set of ACPs depicted in Figure 2, one of the reachable paths it discovers (step 5, section C), takes the user along a sequence of activities, beginning at statement R.4.1.5.5 (start point) and terminating at statement R.3.1.1.7.a.i (end point). This path allows her to create another ordinary user's account.

$R.3.1.1.7.a.i[user_{admin} : create : account_{user}]_{cap} |$

We now complete step five of the algorithm by combining the subject of the start point (R.4.1.5.5) with the action and object of the end point (R.3.1.1.7.a.i) to create the temporary ACP,

$[user : create : account_{user}]_{cap} |$

The capability represented by this ACP however, violates constraint R.4.2.1.2 (not shown in Figure 2 but associated with ACP R.4.2.1.2.i) which states,

$R.4.2.1.2[user : create : account_{user}]_{con} |$

All the steps involved in completing the activity are legitimate, yet the outcome violates the policy. This is a loophole and therefore constitutes a potential vulnerability in the system. Figure 3 is a replica of the digraph shown in Figure 2 with the identified loophole highlighted.

IV. THE SECRET

SECRET is designed to be applied towards the end of the requirements engineering phase, after the use of existing approaches such as misuse cases. Its main features are

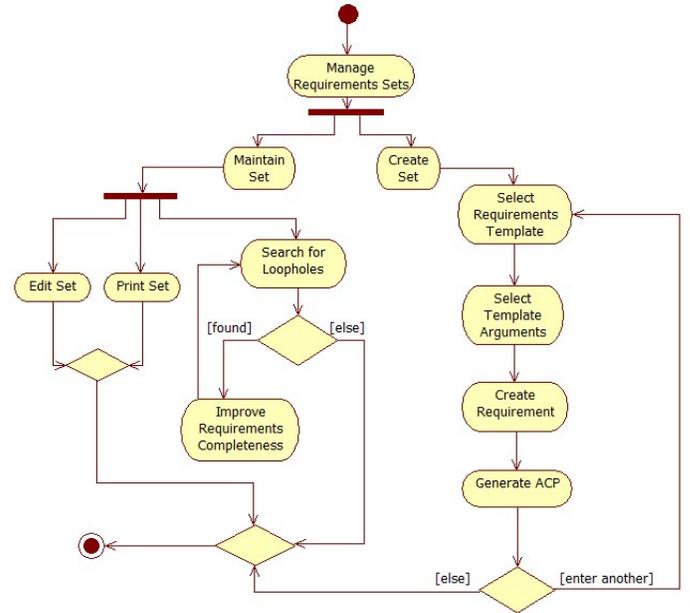


Fig. 4. Overview of the SECRET

described below:

- 1) It permits the entry of requirements. The Volere [22] requirements shell and boilerplates [23] are used
- 2) It extracts the required data and creates properly formatted ACPs
- 3) It permits the maintenance of sets of ACPs: save to a file, print, sort and load from a file
- 4) It permits the maintenance of the ISD table of template arguments and imposed security dependencies
- 5) It automatically generates ACPs based on imposed security dependencies
- 6) It identifies policy breaches using the loophole algorithm
- 7) It generates a basic requirements document using an ACP set, the Volere requirements shell and boilerplates

Figure 4 depicts the main set of activities that an engineer can perform using the SECRET, with the exception of items 4 and 7 listed above.

Figure 5 depicts the architecture of the SECRET's primary functions. The function `analyzeISD()` identifies omitted security related requirements based on the ISD table stored in one of the prototype's SQLite databases. It incorporates the use of four functions that each loop through the set of ACPs and check for omissions related to actions, objects, action and object clarifiers. The four functions themselves call upon two others. `relReqExist()` identifies requirements statements that are unrecorded, but must be included, based on ISDs contained in the tool's ISD table. `genDR()` generates the requirement statements that must be included, but were unrecorded, using templates selected from the

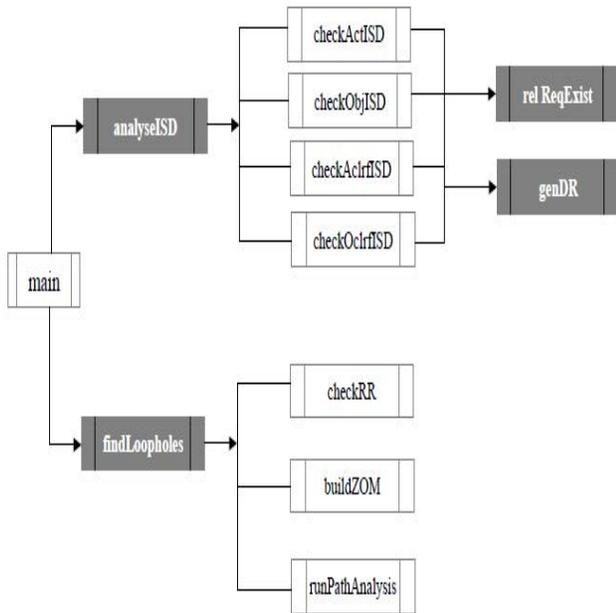


Fig. 5. Architecture of the SECRET's primary functions

tool's collection. The function `analyseISD()` successfully completes when no new requirements statements have been generated. It is these functions that are responsible for detecting the ISD on clarifier 'write' and generating the ACP statement 2.1.5.1 in section 3B.

The function `findLoopHoles()` governs the application of the loophole analysis by executing (among others) the following processes:

- verifies that all successor requirement entries are valid. A successor is deemed invalid if a corresponding ACP does not exist. `findLoopHoles()` cannot continue until all successor requirement entries have been validated.
- builds the binary, $n \times n$ matrix M (where $n=|D|$) using multi-dimensional arrays, and makes a copy of M . This copy is used to complete the transitive closure of M , M' .
- performs the Floyd-Warshall algorithm to complete the transitive closure of R .

`findLoopHoles()` then compares the arrays used to represent M and M' . When a difference is found, the bit-wise XOR is performed on the arrays, temporary ACPs are built, and the function loops through an array of ACPs expressed as constraints. The function then compares the temporary ACPs with these constraints and displays any matches that are found as potential flaws. These matches correspond to loopholes or vulnerabilities that exist in the system's design.

V. CONCLUSION AND FUTURE WORK

Although preliminary, we believe the results are promising. Loopholes are unknown, reachable paths that would exist if a system were to be developed in accordance with the specification document in the form prior to the loophole analysis. The analysis did not include nor require the motives, resources and skills of an attacker, or possible threats to the system to be postulated. Its foundation is based on the statement of policy, expressed as a ACP.

SECRET, although tested using documents between 80-150 pages, has not been tested with documents that detail large and very large systems. This is primarily due to the unavailability of such documents for our research due to the reluctance of organisations to disclose such information.

We are pursuing a number of enhancements to the SECRET. For example, we would like to provide the ability to step through loopholes, represented as digraphs, to visualise the points along the paths that raise concerns. This will provide an engineer with the ability to develop appropriate countermeasures with minimal effect on the intended system's features.

Perhaps the most important feature to be included will be the SECRET's security assurance rating of proposed systems. This rating will assert that the requirements engineers of a proposed system have satisfactorily considered and addressed particular classes of vulnerabilities during the design process. The rating, however, will only be applicable if the system is developed in strict accordance with the specification, after the SECRET's analyses and modifications have been performed and made to the specification.

REFERENCES

- [1] G. Sindre and A. Opdahl, "Eliciting security requirements by misuse cases," in *Proc. of technology of object oriented languages and systems*. IEEE, 2000, pp. 120–131.
- [2] R. De Landtsheer and A. van Lamsweerde, "Reasoning about confidentiality at requirements engineering time," in *ESEC/FSE-13: Proceedings of the 10th european software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 41–49.
- [3] C. B. Haley, J. D. Moffett, R. C. Laney, and B. Nuseibeh, "A framework for security requirements engineering," in *SESS '06: Proceedings of the 2006 international workshop on software engineering for secure systems*. New York, NY, USA: ACM, 2006, pp. 35–42.
- [4] J. McDermott and C. Fox, "Using abuse case models for security requirements analysis," in *Proceedings of Computer security applications conference*. IEEE Computer Society, 1999, pp. 55–64.
- [5] L. Liu, E. Yu, and J. Mylopoulos, "Security and privacy requirements analysis within a social setting," in *RE '03: Proceedings of the 11th IEEE international conference on requirements engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 151.
- [6] A. J. A. Wang, "Information security models and metrics," in *ACM-SE 43: Proceedings of the 43rd annual southeast regional conference*. New York, NY, USA: ACM, 2005, pp. 178–184.
- [7] J. Jurjens, "Umlsec: extending uml for secure systems development," in *Proc. of the 5th international conference on the unified modeling language*. London, UK: Springer-Verlag, 2002, pp. 412–425.

- [8] N. Mead, "Identifying security requirements using the security quality requirements engineering (square) method," in *Integrating security and software engineering: advances and future vision*, H. Mouratidis and P. Giorgini, Eds. IGI Global, 2007, ch. 3, pp. 44–69.
- [9] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM, 2000, pp. 5–19.
- [10] M. Hafiz, P. Adamczyk, and R. Johnson, "Organizing security patterns," *IEEE software*, vol. 24, no. 4, pp. 52–60, 2007.
- [11] MITRE Corporation, "Common weakness enumeration," 2008, [Accessed January 2009]. [Online]. Available: <http://cwe.mitre.org>
- [12] P. T. Devanbu and S. Stubblebine, "Software engineering for security: a roadmap," in *ICSE '00: Proceedings of the conference on the future of software engineering*. New York, NY, USA: ACM, 2000, pp. 227–239.
- [13] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, "Non-functional requirements in software engineering," *International Series in Software Engineering*, vol. 5, pp. 476–, 1999.
- [14] P. Zatkó, "Psychological security," in *Beautiful security: leading security experts explain how they think*, 1st ed., A. Oram and J. Viega, Eds. O'Reilly Media, 2009, ch. 1, pp. 1–20.
- [15] I. Sommerville, *Software engineering*, 9th ed. Addison-Wesley, 2010.
- [16] J. Routh, "Forcing firms to focus: is secure software in your future?" in *Beautiful security: leading security experts explain how they think*, 1st ed., A. Oram and J. Viega, Eds. O'Reilly Media, 2009, ch. 11, pp. 183–197.
- [17] C. Busby-Earle and E. K. Mugisa, "Metadata for boilerplate placement values for secure software development using derived requirements," in *Proceedings of the 13th IASTED international conference on software engineering and applications (SEA 2009)*, Cambridge, Massachusetts, 2009, pp. 196–201.
- [18] F. Piessens, "A taxonomy (with examples) of cases of software vulnerabilities in internet software," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW346, 2002.
- [19] C. Busby-Earle and E. K. Mugisa, "Towards writing secure software requirements," in *Proceedings of the IASTED international conference on software engineering (SE 2009)*, Innsbruck, Austria, 2009, pp. 101–105.
- [20] K. E. Iverson, "Notation as a tool of thought," *Communications of the ACM*, vol. 23, no. 8, pp. 444–465, 1980.
- [21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge, Massachusetts: MIT Press, 2009, pp. 693–698.
- [22] J. Robertson and S. Robertson, "Volere requirements specification template," 2010, [Accessed February 2010]. [Online]. Available: <http://www.volere.co.uk/template.htm>
- [23] E. Hull, K. Jackson, and J. Dick, *Requirements engineering*, 2nd ed. Springer, 2005.