Cyberpatterns 2012
Proceedings of the First International Workshop on Cyberpatterns: Unifying Design Patterns with Security, Attack and Forensic Patterns

9-10 July 2012
The Cosener’s House, Abingdon, UK

Organized and Sponsored by
Oxford Brookes University

Sponsored by
SOPHOS

In association with
BCS Information Security Specialist Group
BCS Formal Aspects of Computing Specialist Group
BCS Cybercrime Forensics Specialist Group
Welcome Message

Cyberspace is increasingly important to people in their everyday lives for purchasing goods on the Internet, to energy supply increasingly managed remotely using Internet protocols. Unfortunately, this dependence makes us susceptible to attacks from nation states, terrorists, criminals and hactivists. Therefore, we need a better understanding of cyberspace, for which patterns, which are predictable regularities, may help to detect, understand and respond to incidents better. The inspiration for the workshop came from the existing work on formalising design patterns applied to cybersecurity, but we also need to understand the many other types of patterns that arise in cyberspace.

We received 26 papers from several continents from which we chose 19 for presentation. We are lucky to have Sean Barnum who is the technical lead for the cyber observables classification (CybOX) and previously for attack patterns (CAPEC), and Kevin Lano who has an extensive background in formal methods and is particularly interested now in formalising UML.

Although this is an academic conference, we have also gained significant interest from industry. We are grateful to our main sponsor Sophos, along with our other supporters listed on the cover page of the proceedings. We would also like to thank the Program Committee for their hard work at short notice.

We are pleased to welcome you to the inaugural cyberpatterns workshop. We hope there will be lively discussion to help establish this nascent field, so that we can plan a roadmap for progression that can be followed up in future workshops.

Clive Blackwell and Ian Bayley
PC Co-Chairs
Programme Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ian Bayley</td>
<td>Oxford Brookes University</td>
<td>UK</td>
</tr>
<tr>
<td>Clive Blackwell</td>
<td>Oxford Brookes University</td>
<td>UK</td>
</tr>
<tr>
<td>Sean Barnum</td>
<td>MITRE, USA</td>
<td>USA</td>
</tr>
<tr>
<td>Bill Buchanan</td>
<td>Edinburgh Napier University</td>
<td>UK</td>
</tr>
<tr>
<td>Tony Clark</td>
<td>Middlesex University</td>
<td>UK</td>
</tr>
<tr>
<td>Nathan Clarke</td>
<td>University of Plymouth</td>
<td>UK</td>
</tr>
<tr>
<td>David Duce</td>
<td>Oxford Brookes University</td>
<td>UK</td>
</tr>
<tr>
<td>Denis Edgar-Nevill</td>
<td>Canterbury Christ Church University</td>
<td>UK</td>
</tr>
<tr>
<td>Steven Furnell</td>
<td>University of Plymouth</td>
<td>UK</td>
</tr>
<tr>
<td>Les Hatton</td>
<td>Kingston University</td>
<td>UK</td>
</tr>
<tr>
<td>James Heather</td>
<td>University of Surrey</td>
<td>UK</td>
</tr>
<tr>
<td>Peter Komisarczuk</td>
<td>Thames Valley University</td>
<td>UK</td>
</tr>
<tr>
<td>Harjinder Lallie</td>
<td>University of Warwick</td>
<td>UK</td>
</tr>
<tr>
<td>Kevin Lano</td>
<td>King's College London</td>
<td>UK</td>
</tr>
<tr>
<td>Alessandro Linari</td>
<td>Nominet, UK</td>
<td>UK</td>
</tr>
<tr>
<td>Shaoying Liu</td>
<td>Hosei University, Japan</td>
<td>Japan</td>
</tr>
<tr>
<td>Carsten Maple</td>
<td>University of Bedfordshire</td>
<td>UK</td>
</tr>
<tr>
<td>Madjid Merabti</td>
<td>Liverpool John Moores University</td>
<td>UK</td>
</tr>
<tr>
<td>Faye Mitchell</td>
<td>Oxford Brookes University</td>
<td>UK</td>
</tr>
<tr>
<td>Gareth Niblett</td>
<td>BCS Information Security Specialist Group, UK</td>
<td>UK</td>
</tr>
<tr>
<td>Richard Overill</td>
<td>King's College London</td>
<td>UK</td>
</tr>
<tr>
<td>Alberto Schaeffer-Filho</td>
<td>Lancaster University</td>
<td>UK</td>
</tr>
<tr>
<td>Jawed Siddiqi</td>
<td>Sheffield Hallam University</td>
<td>UK</td>
</tr>
<tr>
<td>Dianxiang Xu</td>
<td>Dakota State University</td>
<td>USA</td>
</tr>
<tr>
<td>Hong Zhu</td>
<td>Oxford Brookes University</td>
<td>UK</td>
</tr>
</tbody>
</table>
# Table of Contents

*Welcome Message From the PC Co-Chairs* .................................................................................................................. i  
*List of Programme Committee Members* .............................................................................................................. ii  
*Table of Contents* ...................................................................................................................................................... iii  

Hong Zhu, *Towards A General Theory of Patterns* ................................................................................................. 1  
Ian Bayley, *Challenges For A Formal Framework for Patterns* .................................................................................. 6  
Shamal Faily, John Lyle and Simon Parkin, *Tool-supported Premortems with Attack and Security Patterns* ........ 10  
Alberto E. Schaeffer-Filho, Andreas Mauthe, David Hutchison and Paul Smith, *Management Patterns for Network Resilience: Design and Verification of Policy Configurations* ................................................................. 12  
Muhammad Asim, Bo Zhou, David Llewellyn-Jones, Qi Shi and Madjid Merabti, *Dynamic Monitoring of Composed Services* ................................................................................................................................. 17  
Bo Zhou, David Llewellyn-Jones, David Lamb, Muhammad Asim, Qi Shi and Madjid Merabti, *A Heuristic Approach for Secure Service Composition Adaptation* .............................................................................................................. 39  
Faye Mitchell, *An Overview of Artificial Intelligence based Pattern Matching in a Security and Digital Forensic Context* ........................................................................................................................................ 43  
Renato Amorim and Peter Komisarczuk, *On Partitional Clustering of Malware* ......................................................... 47  
Harjinder Lallie and Nathan Griffiths, *Where has this hard disk been? : Extracting geospatial intelligence from digital storage systems* .................................................................................................................. 52  
Noor-Ul-Hassan Shirazi, Alberto E. Schaeffer-Filho and David Hutchison, *Attack Pattern Recognition through Correlating Cyber Situational Awareness in Computer Networks* .................................................................................. 57  
Paul Kearney, David Sinclair and Sebastian Wagner, *Security Design Patterns in the MASTER Workbench* ............ 62  
David Llewellyn-Jones, Qi Shi and Madjid Merabti, *Extending AOP Principles for the Description of Network Security Patterns* .................................................................................................................................. 67  
Nick Walker, Natalie Coull, Ian Ferguson and Allan Milne, *On the Use of Design Patterns to Capture Memory Corruption Vulnerabilities* ........................................................................................................... 72  
Clive Blackwell, *Using Attack Patterns to Develop a Penetration Testing Framework* ........................................ 77  
Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubrina, Daniel Bilar and Michael Locasto, *Composition Patterns of Hacking* ....................................................................................................................... 80  

*Author index* ................................................................................................................................................................. 86
Towards A General Theory of Patterns

Hong Zhu
Department of Computing and Communication Technologies
Oxford Brookes University, Oxford, OX33 1HX, UK
Email: hzhu@brookes.ac.uk

Abstract -- As knowledge of solutions to recurring design problems, a large number of software design patterns (DP) have been identified, catalogued, and formalized in the past decades. Tools have been developed to support the application and recognition of patterns. However, although the notions of pattern in different subject domains carry a great deal of similarity, we are in lack of a general theory that applies to all types of design patterns. This paper is based on our previous work on formalization of OO DPs and an algebra of pattern compositions. We propose a generalization of the approach so that it can be applied to other types of DPs. In particular, a pattern is defined as a set of points in a design space that satisfy certain conditions. Each condition specifies a property of the instances of the pattern in a certain view of the design space. The patterns can then be composed and instantiated through applications of operators defined on patterns. The paper demonstrates the feasibility of the proposed approach by examples of patterns of enterprise security architecture.

Keywords – design space; design pattern; enterprise security systems; formal methods.

I. INTRODUCTION

Since 1980s, much work has been reported in the literature on the patterns of OO software designs. Here, a design pattern (DP) is a piece of codified knowledge of design solutions to recurring design problems. A pattern-oriented design methodology has been advanced by the identification and catalogue of patterns, the formalization of them and the development of techniques and tools for formal reasoning about patterns and automating pattern-oriented design and code recovery. Its success in improving OO design has also fostered research on patterns of other aspects of software design, such as interface, architecture and fault tolerant designs. The notion of patterns has also been extended to other phases of software lifecycle, such as analysis patterns in requirements analysis, architectural patterns in software architectural design, process patterns in software process modelling, test patterns in software testing, etc.

In a more general context, the notion of pattern has been investigated in many subject areas of computer science. In particular, security patterns [1, 2] and attack patterns have been identified and catalogued in the study of computer security. However, although the notions of patterns in different subject areas carry a great deal of similarity, we are in lack of a general theory that applies to all types of patterns.

In this paper, we propose an approach to generalize our previous work on the formalization of OO DPs and algebra of pattern compositions and instantiations. We will also explore the applicability of the general theory to security and identify the new problems in the study of security patterns.

II. RELATED WORKS

A. OO Design Patterns

In the past decade, several formalisms for formally specifying OO DPs have been advanced [3]. In spite of differences in these formalisms, the basic underlying ideas are quite similar. That is, patterns are specified by constraints on what are its valid instances via defining their structural features and sometimes their behavioural features too. The structural constraints are typically assertions that certain types of components exist and have a certain configuration of the structure. The behavioural constraints, on the other hand, detail the temporal order of messages exchanged between the components.

Therefore, in general, a DP \( P \) can be defined abstractly as a tuple \( \langle V, Pr_S, Pr_D \rangle \), where \( V = \{ v_1 : T_1 , \ldots , v_n : T_n \} \) declares the components in the pattern, while \( Pr_S \) and \( Pr_D \) are predicates that specify the structural and behavioural features of the pattern, respectively. Here, \( v_i \)'s in \( V \) are variables that range over the type \( T_i \) of software elements, such as class, method, and attribute. The predicates are constructed from primitive predicates either manually defined, or systematically induced from the meta-model of software design models [4]. The semantics of a specification is the ground formula \( \exists V. (Pr_s \land Pr_d) \).

The notion of pattern conformance, i.e., a concrete design \( D \) conforms to a pattern \( P \), or \( D \) is an instance of \( P \), can be formally defined as logic entailment \( D \models Pr \) (i.e., the statement \( \exists V. Pr \) is true on \( D \) ), where \( Pr = Pr_S \land Pr_D \) and we write \( D = P \). Consequently, for patterns \( P_i, i = 1,2 \), \( \exists V_1. Pr_1 \models \exists V_2. Pr_2 \) means pattern \( P_1 \) is a specialization of pattern \( P_2 \) and we have that for all designs \( D \), \( D = P_1 \) implies that \( D = P_2 \). In other words, reasoning about the specialization relation between patterns and the conformation of designs to patterns can be performed in formal logics.

In [5], we have proposed the following operators on DPs for pattern composition and instantiation.

- **Restriction** \( P[C] \): to impose an additional constraint \( C \) to pattern \( P \);
- **Superposition** \( P_1 \ast P_2 \): to require the design to conform to both pattern \( P_1 \) and \( P_2 \);
- **Generalisation** \( P[x] \): to allow an element \( x \) in pattern \( P \) to become a set of elements of the same type of \( x \);
- **Flatten** \( P[x] \): to enforce a set of elements of pattern \( P \) to be a singleton;
- **Lift** \( P \{x\} \): to duplicate the number of instances of pattern \( P \) in such a way that the set of components in each copy satisfies the relationship as in \( P \) and the copies are configured in the way that element \( x \) serves as the primary key as in a rela-
tional database.

- **Extension** \( P(\forall C \land C) \): to add components in \( V \) into \( P \) and connect them to the existing components of \( P \) as specified by predicate \( C \).

Using these operators, pattern oriented design decisions can be formally represented [6]. A complete set of algebraic laws that these operators obey has also been established so that the result of design decisions can be worked out formally and automatically. Moreover, with the algebraic laws, the equivalence between different pattern expressions can be proven formally and automatically through a normalization process. For example, we can prove that the equation \( P(\forall |X| = 1) \equiv P \cup X \) holds for all patterns \( P \).

### B. Design Space

Generally speaking, a design space for a particular subject area is a space in which design decisions can be made. Each concrete design in the domain is a point in this space. Understanding the structure of a design space of a particular domain plays a significant role in software design [7]. Three approaches to represent design spaces have been advanced in software engineering research:

- **Multi-dimensional discrete Cartesian space**, where each dimension represents a design decision and its values are the choices of the decision.
- **Hierarchical structure**: where nodes in a tree represent a design decision and alternative values of the decision are the branches, which could also be dependent design sub-decisions [8].
- **Instance list**: where a number of representative instances are listed with their design decisions.

In the General Design Theory (GDT) proposed by Yoshikawa [9, 10], a design space is divided into two views: one for the observable (structural) features of the artefacts, and the other for functional properties. These two views are linked together by the instances in the domain. These instances show how combinations of structural properties are associated to the combinations of functional properties. These two views are regarded as topological spaces and the links as continuous mappings between them. By doing so, two types of design problems can be solved automatically.

- **Synthesis problem** is to find a set of the structural features as a solution that has certain functional features that are given as design requirements.
- **Analysis problem** is to find out the functional properties from an object’s structural properties.

The existing work on OO DPs can be understood in the GDT very well, which also provides a theoretical foundation for the approach proposed in this paper. However, existing approaches to the representation of design spaces cannot deal with the complexity of software design satisfactorily. Thus, we propose to use meta-modelling.

### C. Meta-Modelling

Meta-modelling is to define a set of models that have certain structural and/or behavioural features by means of modelling. It is the approach that OMG defines UML and model-driven architecture [11]. A meta-model can be in a graphic notation such as UML’s class diagram, or in text format, such as GEBNF, which stands for graphic extension of BNF [12].

In GEBNF approach, meta-modelling is performed by defining the abstract syntax of a modelling language in BNF-like meta-notation and formally specifying the constraints on models in a formal logic language induced from the syntax definition. Formal reasoning about meta-models can be supported by automatic or interactive inference engines. Transformation of models can be specified as mappings and relations between GEBNF syntax definitions together with translations between the predicate logic formulas.

In GEBNF, the abstract syntax of a modelling language is a 4-tuple \(< R, N, T, S >\), where \( N \) is a finite set of non-terminal symbols, and \( T \) is a finite set of terminal symbols. Each terminal symbol, such as String, represents a set of atomic elements that may occur in a model. \( R \in N \) is the root symbol and \( S \) is a finite set of syntax rules. Each syntax rule can be in one of the following two forms.

\[
Y ::= X_1 | X_2 | \cdots | X_n \\
Y ::= E_1 : f_1 , E_2 : f_2 , \cdots , E_n : f_n
\]

(1) \[(2)\]

where \( Y \in N \), \( X_i \in T \cup N \), \( f_i \)'s are field names, and \( E_i \)'s are syntax expressions, which are inductively defined as follows:

- \( C \) is a basic syntax expression, if \( C \) is a literal instance of a terminal symbol, such as a string.
- \( X \) is a basic syntax expression, if \( X \in T \cup N \).
- \( X @ Z f \) is a basic syntax expression, if \( X, Z \in N \), and \( f \) is a field name in the definition of \( Z \), and \( X \) is the type of \( f \) field in \( Z \)’s definition. The non-terminal symbol \( X \) is called a referential occurrence.
- \( E^* \), \( E^+ \) and \([E]\) are syntax expressions, if \( E \) is a basic syntax expression.

The meaning of the above meta-notation is informally explained in Table 1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X^* )</td>
<td>A set of elements of type ( X ).</td>
</tr>
<tr>
<td>( X^+ )</td>
<td>A non-empty set of elements of type ( X ).</td>
</tr>
<tr>
<td>( [X] )</td>
<td>An optional element of type ( X ).</td>
</tr>
<tr>
<td>( X@Zf )</td>
<td>A reference to an existing element of type ( X ) in field ( f ) of an element of type ( Z ).</td>
</tr>
</tbody>
</table>

Informally, each terminal and non-terminal symbol denotes a type of elements that may occur in a model. Each terminal symbol denotes a set of predefined basic elements. For example, the terminal symbol String denotes the set of strings of characters. Non-terminal symbols denote the constructors of the modelling language. The elements of the root symbol are the models of the language.

If a non-terminal symbol \( Y \) is defined in the form (1), it means that an element of type \( Y \) can be an element of type \( X_i \), where \( 1 \leq i \leq n \).

If a non-terminal symbol \( Y \) is defined in the form (2), then, \( Y \) denotes a type of elements that each consists of \( n \) elements of type \( X_1 , \ldots , X_n \), respectively. The \( k \)’th element in the tuple can be accessed through the field name \( f_k \), which is a function symbol of type \( Y \rightarrow X_k \). That is, if \( a \) is an element of type \( Y \),
we write $a, f_k$ for the $k$’th element of $a$.

Given a well-defined GEBNF syntax $G = \langle R, N, T, S \rangle$ of a modelling language $L$, we write $\text{Fun}(G)$ to denote the set of function symbols derived from the syntax rules. From $\text{Fun}(G)$, a predicate logic language can be defined as usual (C.f. [13]) using variables, relations and operators on sets, relations and operators on types denoted by terminal and non-terminal symbols, equality and logic connectives or $\vee$, and $\wedge$, $\neg$, implication $\rightarrow$ and equivalent $\equiv$, and quantifiers for all $\forall$ and exists $\exists$.

### III. THE PROPOSED APPROACH

#### A. Overview

The proposed approach consists of the following aspects.

- **Definition of design space.**

  We will use GEBNF-like meta-notation to define a meta-model as the design space. The meta-model will define a number of views. In each view, the meta-model will define a number of types of component elements in the subject domain and relations and properties of the elements.

- **Specification of patterns in a design space.**

  The patterns in a design space can then be specified formally using the induced predicate logic in the same way as we define OO DPs. That is, each pattern is defined by a predicate in the induced predicate logic language.

  Patterns can also be defined as compositions and instantiations of existing patterns by applying the operators on patterns defined in [5]. We believe that the algebraic laws proved in [6] should also hold for such design spaces. Therefore, the proofs of properties of patterns can be performed in the same way as in OO design patterns.

#### B. Definition of Design Spaces

We represent a design space in the following form.

```
DESIGN SPACE <Name>;
  <Element type definitions>;
  <View definitions>
END <Name>
```

An element type definition is in the form of GEBNF formula (1). For example, the following is the definition of elements in an object oriented design.

```
DESIGN SPACE OODesign;
  TYPE
    Class ::= name: String, attrs: Property*, ops: Operation*;
    Property ::= name: String, type: Type;
    Operation ::= name: String, params: Parameter*;
    Parameter ::= name: String, type: Type;
  VIEW ...
END OODesign.
```

A view defines a set of properties of the element types and relationships between them together with some constraints.

For example, the following is the structural view of OO designs at class level. The constraint states that inheritance is not allowed to be in cycles.

```
VIEW Structure;
BEGIN
  PROPERTY
    Features:
      [Class | Operation | Property] ->
      [Abstract, Leaf, Public, Private, Static, Query, New]*;
    Direction:
      Parameter -> [In, Out, InOut, Return];
  RELATION
    association, inherits, composite, aggregate: Class x Class;
  CONSTRAINT
    FOR ALL c, d : Class THAT
    NOT (inherits(c,d) AND inherits(d,c)).
END Structure;
```

A view may also contain additional element types. For example, the behavioural view of OO design contains new types of elements such as messages, lifelines, and execution occurrences, frameworks.

#### C. Specification of Patterns

A pattern can be defined in two ways. The first is to define a pattern as a set of points in a design space in the following form.

```
PATTERN <Name> OF <Design space name>;
  COMPONENT <Var>: <TypeExp>*;
  CONSTRAINT [IN <View name> VIEW: <Predicate>]*
END <Name>
```

For example, the Composite pattern in the Gang-of-Four catalogue can be defined as follows.

```
PATTERN Composite OF OODesign;
  COMPONENT leaves: SET Of Class;
  component, composite: Class;
  CONSTRAINT
    inherits{composite, component};
    composite{component, composite};
    FOR ALL c IN leaves THAT inherits{c, components};
  component.features = {abstract};
END Behaviour VIEW ...
```

END Composite.

The second way is to define a pattern as a composition or instance of other patterns by applying the pattern composition operators to existing ones. For example, the following defines a generalised version of the Composite pattern.
IV. APPLICATION TO SECURITY DESIGN PATTERNS

In this section, we apply the proposed approach to security design patterns to demonstrate the style of design space definition and pattern specification in the proposed approach.

A. The Design Space of Security Systems

Computer and network security relies on a wide range of issues and various levels. Here, as an example, we focus on the logic and context level of enterprise architecture. In this case, we can model security systems in box diagrams [14]. A box diagram consists of a number of boxes and arrows. Each box represents a subsystem or entity of the system. Each arrow represents a channel of information flow or interaction between subsystems. For the sake of space, we will only define the structural view of the design space. The dynamic view of system’s behaviour will be omitted.

B. Security System Design Patterns

Now, we demonstrate that security system design patterns can be design with a number of special components that fully fill various security specific functions, such as encryption and decryption.

Figure 1 shows the architecture of an indirect in-line authentication architecture, where AI stands for authentication information.

**Figure 1. Indirect in-line authentication architecture**

This architecture can be represented as follows.

**PATTERN** Indirect In-Line-Authentication IN SecuritySystem;
**COMPONENT**
Claimant, TrustedThirdParty, Verifier: Subsystem;

**CLAIM**
ClaimAl, VerifyAl, ClaimAl2: Subsystem
ClaimAl2VerifyAl, VerifyAl2ClaimAl2: InfoFlow;
ClaimAl2Verifier: InfoFlow;

**CONSTRAINT**
ClaimAI is-a-part-of Claimant;
VerifyAI is-a-part-of TrustedThirdParty;
ClaimAI2 is-a-part-of TrustedThirdParty;
ClaimAl2VerifyAI from = ClaimAI;
ClaimAI2VerifyAI to = VerifyAI;
VerifyAI2ClaimAI2.from = VerifyAI;
VerifyAI2ClaimAI2.to = VerifyAI;
ClaimAI2Verifier from = ClaimAI2;
ClaimAI2Verifier.to = Verifier;

**END**

An alternative authentication pattern is online authentication shown in Figure 2.

**Figure 2. Online authentication architecture**

**PATTERN** Online-Authentication IN SecuritySystem;
**COMPONENT**
Claimant, TrustedTP, Verifier: Subsystem;
ClaimAI, AuthorityClaimAI, VerifAI: Subsystem;
AuthorityVerifAI: Subsystem;
ClaimantTrustedTP, VerifierTrustedTP: InfoFlow;
ClaimantVerifier: InfoFlow;

**CONSTRAINT**
ClaimAI is-a-part-of Claimant;
AuthorityClaimAI is-a-part-of TrustedTP;
VerifAI is-a-part-of TrustedTP;
AuthorityVerifAI is-a-part-of Verifier;
...
(* Some constraints are omitted for the sake of space *)

**END**

Another set of examples of security design patterns are encryption and decryption techniques, as shown in Figure 3.

**Figure 3. Encryption and decryption**

**PATTERN** EncryptDecrypt IN SecuritySystem;
**COMPONENT**
encrypt, decrypt, source, ciphered, recovered,
key1, key2: Subsystem;
source2encrypt, encrypt2ciphered, ciphered2decrypt,
decrypt2Recovered, key12encrypt, key22decrypt: InfoFlow;
There are two types of encryption/decryption techniques: symmetric and asymmetric. The former uses the same key in encryption and decryption, while the later uses different keys. Thus, we have two specialisations of the patterns.

PATTERN SymmetricEncrypt in SecuritySystem EQUALS
EncryptDecrypt [key1.content = key2.content] END

PATTERN AsymmetricEncrypt in SecuritySystem EQUALS
EncryptDecrypt [not (key1.content = key2.content)] END

Figure 4 shows a conceptual model of access control subsystem [14]. It is in fact a design pattern for access control in enterprise systems.

Figure 4. Conceptual model of access control system.

PATTERN AccessControl IN SecuritySystem

COMPONENT
Subject, EnforcementFun, DecisionFun, Object, AuditLogs, AccessControlList, SubjectReg: Subsystem;
AccessReq, ApprovedAccessReq, DecisionReq, DecisionResp, WriteAuditRecord, SubjectInfo,
AccessRule: InfoFlow;

CONSTRAINT ...

END.

V. CONCLUSION

In this paper we have proposed an approach to define design spaces so that design patterns in various subject domains can be defined in the same way as we define OO design patterns. We demonstrated the applicability of the proposed approach by examples of security design patterns. However, the structures of security systems have been simplified by representing them in box diagram models. Their dynamic features are omitted. The examples given in this paper are only skeletons. Many obvious constraints have been omitted for the sake of space. Further details must be worked out. There are also a number of other security design patterns can be identified. A case study of them and their composition is worth trying.

Existing research on relationships between DPs has limited to those within the same design space. However, to study patterns in cyberspaces, we need relationships between patterns across different design spaces. In particular, a security DP may be designated to against an attack pattern. They are in different design spaces. Hence, we have the following research questions:

- How to formally define the ‘against’ relationship between such pairs of patterns? And, how to prove a security pattern can successfully prevent all attacks (i.e. instances) of a certain attack pattern?
- Assume that the composition of security DPs (and attack patterns as well) be expressed in the same way as composition of OO DPs. Then, a question is: if a number of security patterns are composed together to enforce the security for an information system, can they prevent attacks of the target attack patterns and their all possible compositions?

ACKNOWLEDGEMENT

The work reported in this paper is funded by the Oxford Brookes University. The author would like to thank Prof. David Duke, Dr. Ian Bayley and Mr. Clive Blackwell for discussions on related topics.

REFERENCES

Challenges For A Formal Framework for Patterns

Ian Bayley

Abstract—This article discusses the possibility of formalising patterns for cybersecurity building on previous successes in formalising design patterns, and focusing on the possibilities of applying the same techniques to attack patterns and security patterns. This would be enhanced by a common framework of patterns.

Index Terms—design patterns, cybersecurity, predicate logic

I. INTRODUCTION

Patterns are general, reusable solutions to commonly occurring problems within a given sphere of computer science. They originated in architecture, were transferred to software design as design patterns, in the so-called Gang of Four (GoF) book [6], and from there, by analogy, they entered the realm of cybersecurity, as attack patterns and security patterns. In all three cases, patterns are presented informally, as natural language text organised under headings. Although even informal documentation has done much to enhance the exchange of knowledge between practitioners, the software engineering community has found that software tools can be written that use formal definitions of design patterns to diagnose patterns instances. They could also assist in the implementation of patterns by “reminding” the user of the necessary rules. So it is natural to ask whether formality in the definition of attack patterns and security patterns can likewise benefit the cybersecurity community. If so, it may even be possible for tools to be kept updated by communicating newly discovered patterns in this way.

We begin by reviewing the concept of software design patterns. We explain one approach to formalisation and then extend it to express the intent of a pattern. We then review the headings of design, attack and security patterns with a view to proposing a common set of headings with a common approach to formalisation.

II. DESIGN PATTERNS

A. Design Patterns as Solutions

The Strategy Pattern, shown here, is used to select an algorithm at runtime.

Each separate implementation of operation $\text{algInt}$ is a separate algorithm. To change the algorithm, we just need to create an object of a different subclass of Strategy and replace the existing object. We can formalise this as a schema, the declaration part of which identifies the components of the diagram and the predicate part of which specifies the constraints on those components. Let $C$ denote the type of classes and let $\text{	extcopyright}$ denote the type of operations.

Strategy $\text{Context, Strategy : C}$
$\text{conInt, algInt : } \text{	extcopyright}$
$\text{ConcreteStrategies : P (C)}$

$\text{Context * Strategy}
\text{conInt } \in \text{Context.operators}$
$(\text{algInt}) = \text{Strategy.operators}$
$\text{algInt.isAbstract}$
$\forall CS \in \text{ConcreteStrategies}$
$CS \rightarrow \text{Strategy} \wedge \neg CS.\text{isAbstract}$
$\text{calls(conInt, algInt)}$

Above, we use the following relations between elements of $C$ and $\text{	extcopyright}$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \rightarrow C'$</td>
<td>class $C$ contains an instance of $C'$</td>
</tr>
<tr>
<td>$C.\text{opers}$</td>
<td>the operations of class $C$</td>
</tr>
<tr>
<td>$C.\text{isAbstract}$</td>
<td>class $C$ is abstract</td>
</tr>
<tr>
<td>$O.\text{isAbstract}$</td>
<td>operation $O$ is abstract</td>
</tr>
<tr>
<td>$C \rightarrow C'$</td>
<td>class $C$ inherits from class $C'$</td>
</tr>
<tr>
<td>calls($O, O'$)</td>
<td>operation $O$ calls operation $O'$</td>
</tr>
</tbody>
</table>

A software system conforms to the pattern if there exist bindings for the variables in the declaration part that make the statements in the predicate part true. Given such a formal definition, a tool can indicate whether a pattern appears in a software system and if so, where it does, in the sense of which variable bindings make the variable bindings true. Here, we have presented design patterns as concrete instantiations of a general solution to software problems.

B. Further benefits of formality

It is also possible to infer consequences from a design. For example, [7] shows how to prove that the abstract methods in the Template Method pattern must be defined in a subclass if that subclass is concrete, and the same argument applies to the similar Strategy pattern shown here. Similarly, it is possible to state formally and prove correct the proposition that one pattern is a generalisation of another eg that the Composite pattern is a generalisation of the Interpreter pattern. Although there are relatively few examples of generalisation within the Gang of Four catalog, it is noteworthy that the concept completely pervades the
presentation of attack patterns and so a formal definition may well prove to be useful.

C. The Intent of Design Patterns

In the above, we have captured the concept of patterns being general solutions, with the concept of instantiation, without expressing the problems for which they are general solutions. Turning back to the diagram of the Strategy pattern, we can see that although the operation \texttt{Strategy algInt} is being called, the operation being run is one of \{\texttt{ConcreteStrategy algInt}, \texttt{ConcreteStrategy algInt}\}. (We use the notation \texttt{C.a} to indicate the operation of an operation \texttt{a} defined in class \texttt{C}.) This set represents the family of algorithms that might be selected. So in general, the intent of the Strategy pattern can be expressed as follows:

\begin{equation}
\text{runsOneOf (Context, conInt, imps(S\texttt{trategy algInt}))}
\end{equation}

Here, for any abstract operation \texttt{a}, let \texttt{imps(a)} be the set of concrete implementations of that operation; since \texttt{algInt} is the only operation in the strategy class there is one for each concrete strategy if we assume that implemented operations are not overridden.

The expression \texttt{imps(a)} can formally be defined as

\begin{equation}
\{C'.a \mid C \neq C' \land C' \rightarrow C \land \neg C'.isAbstract\}
\end{equation}

The tool can be used in a concrete situation as follows. The user defines the intent (s)he wants in the language introduced in the table above augmented with predicates such as \texttt{runsOneOf}. After supplying the relevant operations \texttt{as} in some way, the method call can be added by the tool automatically, together with the relevant classes with those operations distributed among them, by unifying \texttt{as} with \texttt{imps(a)}, providing a valuable conceptual shortcut, and enabling the user to think usefully at a more abstract level than that given by the UML.

III. DESIGN VS ATTACK AND SECURITY PATTERNS

Although it is very much an open problem whether or not attack and security patterns can be formalised in this way, we are working towards a common framework. Such a framework would involve logic expressions of the sort seen above interspersed with headings like those seen in the informal definitions. So there is a need to identify a common set of headings that can be used for all three types of pattern; at the moment different headings are used for each.

A. Design Patterns

Design patterns are general reusable solutions to commonly occurring \textit{software design} problems. Each pattern is given a name and some aliases; the same is true for attack patterns and security patterns.

The problem to be solved, what we call the intent, is described by several headings. The main such heading is Intent, which is a one-or-two-sentence summary short enough to be used for selecting candidate patterns to read in a given situation. However, other headings that describe the problem are Applicability and the first half of Motivation.

Likewise, the solution is described by several headings. The main heading is Structure which consists of a UML-like class diagram. This diagram is generic in its identifier names and intended to be representative of several such solutions. For example, if an inheritance hierarchy is drawn with two subclasses, it is generally understood that three, four or even one subclasses would be acceptable. This is apparent in the diagram for Strategy pattern given above.

Two other headings are Participants and Collaborations, which mostly contain information directly readable or inferable from the diagram. We discussed the possibilities of inferring information from the diagram in the previous section. The information in Consequences is less directly inferable or not inferable at all and some of it restates the intent. Another two important headings are Sample Code and Motivation (again) which give illustrative concrete instances of the pattern defined by the generic diagram in Structure. The heading Known Uses does this too but with less detail.

Clearly, the relation of instantiation between patterns and pattern instances is an extremely important one for understanding patterns, since that is necessary for patterns to be identified in existing code and to be used to create new code. The relationship between the problem and the solution also appears to be important. We discussed both in the previous section. Finally, Implementation gives optional suggestions, and Related Patterns, in spite of its name, recommends different ways of composing the pattern with other patterns to give new patterns.

B. Security Patterns

Security patterns [3] are like design patterns except that the recurring problems being solved are in the sphere of information security. We focus on the 16 patterns categorised as structural patterns, rather than procedural patterns. These tend to focus on matters such as passwords, privileges, and sessions. The Abstract heading is similar to the Intent heading of design patterns. The Problem heading does not give a single concrete example like the Motivation heading of design patterns. Instead it describes various subtypes of the pattern. There is an Examples heading too and it performs a similar role, at a similar level of detail, to the Known Uses heading of design patterns.

The Solution heading has block diagrams, which are intended perhaps more for pictorial illustration than detailed analysis. Components can be deployment nodes (eg client or server), files (eg web pages), data items (eg session key), code modules (eg filter or encryption module). All are related by only one form of connection. Crucially, the diagrams are in any case optional. For some patterns, such as Network Address Blacklist, the heading contains a list of components. For others, such as Password Authentication, it contains a step-by-step recipe.

Unlike with design patterns, the Related Patterns head-
ing describes similar alternative patterns, as opposed to possible combinations of patterns. The Issues and Trade-Offs heading taken together correspond to the Implementation heading of design patterns, though Trade-Offs is more structured.

C. Attack Patterns

Attack patterns [2], in contrast, solve the problems of those wishing to compromise information security. The problem to be solved is given by the Attacker Intent and Motivation sections. The solution is given by the Attack Execution Flow, which presents algorithms using sequence and selection constructs. Examples are given under the Examples-Instances heading. The Related Patterns heading includes specialisations and generalisations of the pattern being documented. Another heading is Attack Pre-requisites, the essential pre-conditions required for the attack to take place.

IV. Commonalities between patterns

It seems likely therefore that the following commonalities between patterns will need to be part of a general theory of patterns.

- Patterns must be instantiatable. It should be possible to produce a clear unambiguous list of actions for a concrete instance given a description of a pattern and it should be possible to easily identify to which pattern a concrete instance belongs.
- There should be a generalisation relationship between patterns such that one pattern is a subclass or superclass of the other.
- It should be possible to infer consequences from a pattern solution.
- In particular, it should be possible to state the problem that the pattern is intended to solve and prove that the pattern solves the problem. In the case of the intent of Strategy above, the concept runsOneOf is defined informally, but perhaps an independent more formal definition is possible.
- It should be possible to compose two patterns to form a new pattern that solves a greater problem than either of the two constituent patterns. Much work has already been done on defining compositions of design patterns, and it is possible that the same might be done for attack and security patterns.
- Finally, although this is only implicit in each type of pattern, each pattern can be seen as a modification of an existing system to achieve a certain result, be it attacking a system with an attack pattern, defending a system with a security pattern, or augmenting a software system with access to a particular class or method, as with design patterns. We see this with the extra classes added to get from the intent of the Strategy pattern to the solution.

A. Formalisation of Design Patterns

Aspects of such a general theory of patterns have already been realised for design patterns as already discussed. However, three further ideas are of possible interest.

Work by Jason Smith [5] has demonstrated that some design patterns can be thought of as being composed of more elementary patterns. For example, Decorator is shown to consist solely of three more elementary patterns chained together. This echoes the original architectural patterns by Alexander, which as [4] points out, are documented in terms of both the smaller (in terms of physical space) sub-patterns which the pattern is made up of, and the larger patterns of which it can form a part.

Also, Amnon Eden [1] describes design patterns in terms of recurring concepts such as tribes of related classes sharing the same methods all in the same hierarchy. It would be very useful to identify such similar recurring concepts in attack and security patterns, and then to work on formalising them or make them atomic concepts in a new theory. Candidates in the case of security patterns would include passwords, the data resources they are needed to access, the hardware resources on which they reside etc.

Finally, there appears to be some counterbalancing relationship between some security patterns and some attack patterns. For example, the security pattern of Password Lock Out would appear to counterbalance (or in other words, defeat) an attack pattern in which a password is broken by exhaustively trying all possible combinations.

B. Potential benefits of formalisation

It seems possible that the activities listed above that are supported by tools for design patterns (or can be supported where the tools have not yet been produced) could also be supported for attack and security patterns. For example, penetration testing could be assisted if attacks could be automated from a generic description in the form of a formalised attack pattern. Conversely, attacks could be recognised in real-time given a library of attack patterns, and patterns could be exchanged more easily given a common formalisation. Furthermore, composite attack patterns could be composed more easily to generate an attack, or decomposed to detect a complex attack. Analogues of all these are currently possible for design patterns.

Furthermore, although this can be achieved without formality, a formalisation of generalisation could assist in “tidying up” existing knowledge. All three types of patterns document variants but they do so in different ways. Attack patterns are most explicit, with headings for generalisation and specialisation. Security patterns achieve this with a combination of the Issues, Trade-offs and Related Patterns headings. Design patterns do this with the Implementation headings.

C. Barriers to successful formalisation

Design patterns are formalised as characteristic predicates on a model of a software system. In other words, the predicate is true of the model if and only if the design pattern is to be found in the system. A single model of software systems can be used for all design patterns in the GoF book. Typically this predicate, asserts the existence
of classes and methods with certain properties. Usually the definitions of methods are not constrained except to dictate that one method must call another method. In a sense, formalisation of design patterns works because it is possible to produce a single model of software systems with a suitable level of abstraction.

Somehow, a similar model, containing all relevant information but not too much fine detail, should be devised for attack patterns and security patterns. It may be possible to start with different piecemeal models for each of the different categories of attack patterns but only a unified model will allow arbitrary attack patterns to be composed.

V. Conclusion

A unified model of design, attack and security patterns facilitating formal reasoning and tool support in the manner proposed above will need to have the following elements:

1. a definition of patterns as a modification of a system, together with definitions of
   (a) composition
   (b) instantiation
   (c) intent satisfaction
2. for each of attack, security and design patterns:
   (a) a single model suitable for expressing all solutions for that class of patterns
   (b) a single model similarly suitable for expressing all problems for that class of patterns; this could be an extension of the model for solutions, as proposed above for design patterns
   (c) a set of base patterns that do not derive from (ie do not specialise) any other patterns
   (d) a set of patterns derived from the base patterns
3. each pattern should be listed without alternatives since these should be factored out into different patterns, and each pattern should consist of:
   (a) name and aliases
   (b) a list of the patterns from which they derive
   (c) a list of the patterns that derive from it (for convenience of reference)
   (d) a problem defined in predicate logic, stated in terms of the problems of patterns from which the pattern derives, if it is a base pattern
   (e) the condition, defined in predicate logic, in which the pattern is applicable
   (f) a solution defined in predicate logic, stated in terms of the solutions of patterns from which the pattern derives, if it is base pattern
   (g) a proof that the solution solves the problem
   (h) a list of examples, written as instantiations of any variables in the formalisations
   (i) a list of pattern compositions involving that pattern that have been found to be useful.

Also, ideally the pattern should be defined in terms of a modification of the existing system. Although this representation may seem too abstract for human readers, tools can transform it into representations more suitable for human readers by, for example, flattening the inheritance hierarchy and instantiating the solutions and problems to illustrate examples.

This work can then perhaps be extended to procedural security patterns, and HCI patterns.

References

[2] Mitre Corporation Common Attack Pattern Enumeration and Classification Available at: capec.mitre.org
[5] Smith, Jason McC Elemental Design Patterns Addison Wesley 2012
[6] Gamma, Helm, Johnson, Vlissides Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley 1994
Tool-supported Premortems with Attack and Security Patterns

Shamal Faily *, John Lyle * and Simon Parkin †
* Department of Computer Science, University of Oxford. first.last@cs.ox.ac.uk
† School of Computing Science, Newcastle University. s.e.parkin@ncl.ac.uk

Abstract — Security patterns are a useful technique for packaging and applying security knowledge. However, because patterns represent partial knowledge of a problem and solution space, there is little certainty that addressing the consequences of one problem won’t introduce or exacerbate another. In this abstract, we suggest that rather than using patterns exclusively to explore possible solutions to security problems, we should use them to carry out a premortem on why they instead cause problems. We present the approach taken to devise and tool-support such a process using data from the EU FP 7 webinos project.

Index Terms — security pattern; attack pattern; premortem; CAIRIS

I. MOTIVATION

Because security knowledge isn’t always readily available in design situations, there is value in codifying and packaging it. Given both the adversarial nature of security, and the resulting dangers of over or under commensurate treatment of security, it seems useful to package knowledge about attacks as patterns as well. It is surprising, therefore, that, despite an abundance of examples of how security knowledge can be codified as patterns, e.g. [1], and the claim that building attack patterns is evidence of organisational security maturity [2], there is a dearth of work describing the application of attack patterns in security design.

While attempts have been made to characterise attack and misuse patterns, e.g. [3], such patterns may not be effective until we understand the abstractions used by attackers, as well as defenders, to reason about a system. One way of tackling this problem involves getting a better understanding of the attackers themselves. Steps towards this goal are being made using profiling techniques [4], and the reuse of open-source intelligence for building attacker personas [5].

Like patterns in general, attacker representations can only provide a partial representation of an attacker’s knowledge; if these are to act as an impetus for motivating attack patterns then the qualitative data used to build these personas needs to be relevant to the design context. For example, if we develop personas for the possible attackers of an online book-store, there is no certainty that these personas will be equally useful when considering the potential attackers of an electronic voting system. Moreover, given that recent work has shown that the data used to develop personas can be useful for informing secure system design in its own right [6], one might argue that if context specific qualitative data was readily available then we would simply use it to identify criteria for selecting a security pattern, thereby eliminating the need for attacker representations, and attack patterns in general.

To better appreciate the value that attack patterns might have in design, we need to consider security as, what social planners call, a wicked problem [7].

II. PATTERNS AS AN EXPLORATORY TOOL

Security can be considered an example of a wicked problem because we lack clarity about what it means to secure systems, tests for proving a system is secure, and a grasp of all possible solutions for satisfying a specified security problem [8]. Making any design decision has consequences on the underlying system. This makes security patterns useful because pattern templates describe the consequences of their use. This is important because the wicked nature of security means that we may never have the assurances that we would like about a pattern’s efficacy; while a pattern may be one possible solution to a problem, we can never be completely sure that this solution itself doesn’t introduce complications yet to be identified. Nonetheless, applying security patterns remains useful because, as designers, they force us to make value judgements about possible design solutions, and these help us delimit the solution space.

Interestingly, the value associated with applying patterns to delimit the problem space is obtained whether or not they successfully address the problem we had in mind. While it seems paradoxical that we would apply a security pattern knowing that it will fail, the value the failure provides in delimiting the problem space is arguably greater than its success. This is because analysing the failure may lead to more reflection about why the failure occurred so that subsequent candidate solutions can avoid any identified pitfalls. Such an approach is analogous to a premortem. In business scenario planning, these operate on the assumption that a solution has failed; rather than reflecting on what may go wrong with a design, planners instead generate plausible reasons for explaining why a solution has already failed [9]. Although the known structure, motivation, and consequences of security patterns provide some insight into the causes of such a failure, when combined with attack patterns, they allow reflection on the motivations of a perceived attacker, and how his capabilities and motivations lead to an exploit identified in a failed security pattern; this can then be considered in subsequent patterns exploring the same problem. If the mapping between patterns is unclear, the lack
of data also provides clues about what additional evidence is needed before a “cause of death” can be established.

III. A TOOL-SUPPORTED PREMORTUM PROCESS

At the University of Oxford, we have explored how such a premortem process might be tool-supported. Using the EU FP7 webinos project as an exemplar, we have imported project requirements, use cases, personas, and open-source threat data from the OWASP [10] project into the open-source CAIRIS design tool [11]. Using the canonical Design Patterns template prescribed by [12], we concurrently specified security and attack patterns that were relevant to webinos in XML documents; an example of the template used for attack patterns is illustrated in Figure 1. Each element of the security and attack patterns was aligned with elements of the IRIS meta-model [13], upon which CAIRIS was built. Once the patterns were created, we first imported the relevant attack patterns into the tool before introducing a security pattern we wish to analyse into a CAIRIS model. In addition to generating a risk analysis model, such as that illustrated in Figure 2, extensions to CAIRIS were also added to automate an attack resistance analysis. This form of analysis was proposed by McGraw [14] as part of an architectural risk analysis process but, instead of using it to demonstrate the viability of known attacks against a security pattern, we instead used the technique to understand why the security pattern failed to mitigate the attack pattern.

We are currently evaluating both this process and the tool-support by using it to support the design of the security architecture for webinos.

IV. ACKNOWLEDGEMENT

The research described in this paper was funded by the EU FP7 webinos project (FP7-ICT-2009-05 Objective 1.2).
Management Patterns for Network Resilience: Design and Verification of Policy Configurations

Alberto Schaeffer-Filho, Andreas Mauthe and David Hutchison
School of Computing and Communications, InfoLab21, Lancaster University, UK
Email: {asf, andreas, dh}@comp.lancs.ac.uk

Paul Smith
Safety and Security Department
AIT Austrian Institute of Technology
Email: paul.smith@ait.ac.at

Abstract—Computer and communication networks are becoming increasingly critical in supporting business, leisure and daily life in general. Thus, there is a compelling need for resilience to be a key property of networks. The approach we present in this paper is intended to enable the specification of management patterns that describe the dynamic intrusion tolerant behaviour of resilient networks. A management pattern describes a policy-based collaboration between a set of resilience mechanisms used to address a specific type of challenge. Much of the existing work on security patterns has focused only on the static defence aspect of a network. However, dynamic behaviour adds a great deal of complexity to network management, thus making the specification of patterns for this very desirable.

Index Terms—Network resilience, management patterns, policy configurations

I. INTRODUCTION

Computer and communication networks are becoming increasingly critical in supporting business, leisure and daily life in general. There is also an evident increase in the number of cyber attacks and security threats to networked systems. Thus, there is a compelling need for resilience to be a key property of networks. Resilience is the ability of the network to maintain acceptable levels of operation in the face of challenges, such as malicious attacks, operational overload, mis-configurations, or equipment failures [1]. The study of network resilience deals with the dependability aspects of networked systems, but also involves other disciplines more related to network security and Quality of Service (QoS). In this context, the term resilience management encompasses and perhaps supplants some elements of the traditional FCAPS (fault, configuration, accounting, performance, and security) network management functionalities.

The nature of network attacks and challenges in general typically requires the coordinated use of a range of resilience mechanisms across various layers of the protocol stack, across a number of administrative domains, and across heterogeneous infrastructures. Our approach to the resilience problem is framed in the context of a two-phase high-level strategy, called \( D^2R^2 + DR: \text{Defend, Detect, Remediate, Recover} + \text{Diagnose, Refine} \) [1]. The strategy deals not only with the installation and configuration of various defence mechanisms, such as firewalls, but also with the dynamic adaptation of the configuration of a network in response to detected challenges – in effect implementing intrusion tolerance, an aspect of security that is increasingly being understood to be vital.

In order to ensure the resilience of a network, we advocate the systematic design and evaluation of resilience strategies, the careful coordination of various resilience mechanisms for monitoring and control, and also the capture of best practices and the experience of network operators into reusable management patterns [2]. Management patterns define an implementable policy-based collaboration between a set of resilience mechanisms used to address a specific type of challenge. Thus, a set of management patterns can describe the dynamic intrusion tolerant behaviour of resilient networks. Much of the existing work on security patterns has focused only on the static defence aspect of a network [3][4]. However, dynamic behaviour adds a great deal of complexity to network management, thus making the specification of patterns for this very desirable. Management patterns can promote the systematic design and reuse of tested solutions for a range of potential challenges when building network resilience configurations.

The remainder of the paper is structured as follows: Section II describes the background and related work relevant to this paper. Section III outlines the idea of reusable patterns for the specification of resilience strategies and presents an example of a pattern to combat a high-volume traffic challenge. Section IV discusses a number of research issues related to management patterns, namely: the specification of a catalogue of attack- and challenge-specific management patterns, conflict analysis and resolution, and pattern and policy refinement. Finally, Section V presents the concluding remarks.

II. BACKGROUND AND RELATED WORK

We previously defined a management pattern as an abstract resilience configuration for addressing a particular type of network challenge [2]. During run-time, when challenges are observed in the live network, one or more management patterns can be automatically selected and instantiated, in conjunction with specific devices, based on the current availability and capability of resources. The aim is to support the systematic design and reuse of solutions for a range of challenges when building dynamic network resilience configurations. Policy-based management [5] is used to control the operation of mechanisms within a pattern, and how they should be reconfigured in the face of new types of challenges or changes in their operational context.
In [6], a number of application-independent structural constructs were presented for specifying the systematic federation of policy-based systems. These constructs, named architectural styles, are similar in intent to software design patterns [7] in the sense that they provide a set of standard solutions for recurring problems. Design patterns and software architectures can be seen as a set of principal design decisions made during the conceptualisation and development of a system [8]. Software architecture-based approaches typically separate computation (components) from interactions (connectors) to design and specify the overall structure of a system. The benefits brought by this distinction have been widely recognised as a means of structuring software development for distributed systems [9].

Configuring policy-enabled resilience mechanisms can present difficulties, especially when one considers the interaction between mechanisms for detection and remediation. Therefore, we propose the use of software engineering principles and management patterns to assist in the design of network resilience strategies. Although components and connectors do not cater for the adaptive behaviour of policy-based systems, similar principles can be applied for designing and reusing these systems.

III. REUSABLE PATTERNS FOR RESILIENCE STRATEGIES

A management pattern is a policy-based configuration for a set of resilience mechanisms and their relationships. Patterns are used to address a particular type of network challenge. Different challenge types will demand specific sets of mechanisms to monitor features in the network (e.g., current traffic load or alarms generated by an anomaly detection mechanism), and initiate remediation actions to combat anomalous behaviour (e.g., blocking malicious flows or selectively dropping packets). This assumes the existence of policy-driven mechanisms supporting a range of resilience functions in the network.

A. Pattern Specification

Patterns are abstractly specified in terms of roles, to which management functions and policies are associated. Roles in a given pattern represent the types of mechanism that are required to combat a specific challenge, and a given pattern may require roles with specific capabilities in terms of their ability to, e.g., monitor links, capture flows or classify traffic. Thus, a pattern specification consists of:

(a) the types of required mechanisms (represented by roles)
(b) how these mechanisms must interact with each other.

For example, a pattern for combating a flash crowd challenge may include roles such as VMReplicator and Web-ServerMonitor, to replicate virtual machines and monitor Web service activity, respectively, whereas a pattern for addressing a DDoS attack may include roles such as TrafficClassifier and RateLimiter. To these roles, specific mechanism instances are assigned when a pattern is instantiated. Note, the same mechanism instance may be assigned to more than one role. For example, an enhanced router [10] may perform the generation of netflow records and also filter malicious flows. Role assignment provides a way of type-checking [6] mechanism instances in order to verify their suitability for the respective roles.

In addition to roles, a pattern also defines the management relationships between these roles, in terms of the policies that should be loaded and the events that should be exchanged. These relationships are expressed in terms of more primitive architectural styles, which are used as building blocks to connect the roles in the pattern. A catalogue of styles defining common management relationships for federating policy-based systems, such as p2p, event diffusion, hierarchical management, has been presented in [6]. The manner in which roles are connected using such architectural styles prescribes the relationships between the mechanisms used in a pattern.

B. Example Scenario: High-Volume Traffic Challenge

Fig. 1 illustrates part of the textual specification1 of a pattern to combat high-volume traffic challenges, e.g., a DDoS attack. The pattern is parameterised with four roles, which are described in Table I. The relationships between the roles are defined in terms of the policies that must be loaded and the events that must be exchanged. In lines 6-14, styles are used to establish these relationships. In particular, we are setting an event diffusion between IFIPFlowExporter and Classifier (lines 6-8), another event diffusion between AnomalyDetection and RateLimiter (lines 9-11), and a hierarchical policy loading between AnomalyDetection and RateLimiter (line 12-14).

A set of policies that can be loaded into another system is called a mission [12]. Fig. 1 defines two missions, throttling in lines 16-19 (to be loaded into RateLimiter) and detection in lines 21-30 (to be loaded into AnomalyDetection). The former specifies a simple rate limiting policy when a specific IP address is deemed suspicious, and the latter defines what should occur when an anomaly is detected with a certain confidence level (%), in particular flag the anomaly, configure Classifier to use a specific algorithm, and generate an alarm.

1 We use a succinct pseudo syntax but in the current implementation patterns are written in PonderTalk [11] which is more verbose. We also limit the example to the configuration of a small set of mechanisms.

| TABLE I
<table>
<thead>
<tr>
<th>ROLES CONTAINED IN A PATTERN TO COMBAT HIGH-VOLUME TRAFFIC CHALLENGES, SUCH AS A DDoS ATTACK.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnomalyDetection</td>
</tr>
<tr>
<td>IFIPFlowExporter</td>
</tr>
<tr>
<td>RateLimiter</td>
</tr>
<tr>
<td>TrafficClassifier</td>
</tr>
</tbody>
</table>
When instantiated, a pattern will deploy the policy-based configurations across the mechanisms assigned to its roles in order to implement the prescribed interactions. Note, a pattern is instantiated only if the available mechanism instances satisfy the requirements for their roles [2]. This ensures that the policies inside the pattern can be executed by these mechanisms. Patterns facilitate the systematic building of policy-driven configurations, and can also be reused to cater for similar challenges that manifest at different parts of the network, or variations of an attack.

IV. RESEARCH ISSUES

This section discusses a number of research issues related to the notion of reusable management patterns for resilience strategies.

A. Attack- and Challenge-specific Management Patterns

A key research issue is to be able to define management patterns that are effective against a particular challenge. As discussed, an attack- or challenge-specific management pattern can be seen as a “recipe” for addressing a particular type of network challenge. For example, Distributed Denial of Service (DDoS) attacks [13] aim to saturate a target resource, such as a set of network links or servers, with unusually high demands for service. A key challenge for service providers is to be able to determine automatically when such a situation (manifested as large volumes of network traffic) can be attributed to a DDoS attack, with nefarious ends, or a legitimate flash crowd event. This is important because the best means to mitigate these two challenges can be quite different.

Regarding detection and remediation of these challenges, a number of candidate resilience mechanisms can be applied. For example, monitoring sensors, anomaly detection systems, and traffic classifiers can be used to detect and characterise a challenge, and determine whether unusually high volumes of traffic are being observed, e.g., caused by a DDoS attack. Complementary to anomaly detection systems are signature-based intrusion detection systems (IDSes) that attempt to match observed traffic against sets of rules or signatures, which describe attacks. With respect to remediating a DDoS attack, various forms of traffic shaping can be used, from blocking traffic to probabilistic throttling, which can be applied at different protocol levels and to individual network device ports, for example. Moreover, other techniques such as load balancing and virtual service migration could be used to remediate a flash crowd event.

Apart from statically configuring (or deploying) these mechanisms, it is also necessary to describe their relationships to orchestrate dynamic behaviour, e.g., based on output from a classifier, replicate a service because of a flash crowd rather than perform traffic shaping. It is our aim to understand how different malicious attacks and other network challenges manifest and how they can be optimally addressed. This knowledge can then be used to build a catalogue of implementable management patterns that capture best practices on how to configure the necessary resilience mechanisms.

To support the development of a catalogue of management patterns, we have developed a resilience simulator [14] that couples the OMNeT++ network simulator2 and the Ponder2 policy management framework3. The purpose of the simulator is to evaluate the effectiveness of policy-driven resilience strategies that address a given challenge. Strategies that prove to be effective in the simulator can subsequently be promoted to management patterns. At the moment, the promotion of policy-driven strategies to patterns is carried out manually; future work will investigate tools to support this process.

B. Conflict Analysis and Resolution

In complex multi-service networks, determining the existence of conflicting configurations is of critical importance. One of the key aspects that need to be addressed is the formal analysis of these configurations to prevent collateral effects, such as policy conflicts [15]. Formal analysis is necessary to assist in the design of policy-based systems and allow the verification of the correctness of anticipated interactions before these are implemented or policies are deployed.

The categorisation of policy conflicts presented in [15] is particularly relevant. The authors identify a set of primitives that can be used for policy ratification, namely: dominance checks, for determining when a policy is dominated by another, and therefore does not effect the behaviour of the system; potential conflict checks, where two or more policies cannot be applied simultaneously; coverage checks, which can be used to identify if policies have been defined for a certain range of possible cases; and consistent priority assignment, to prioritise

---

Footnotes:
1 http://www.omnetpp.org/
2 http://ponder2.net/
on highUtilisation (link) {
    RateLimiterMO limit (90%);
}

on highServiceUtilisation (service) {
    VMReplicatorMO replicateService (service);
}

Vertical
conflict
Defend Detect Remediate Recover
Strategy
Levels
Network Service Resilience Mechanisms
IDS Link Monitor Rate Limiter Local Manager Classifier Codec Adapter VM Replicator Service Monitor IDS Overlay PathSelector

on classification (f1, value, conf) {
    if ((value == 'DDoS') and (conf <= 0.8)) {
        RateLimiterMO limit (f1.src, f1.dest, 80%);
    }
}

on classification (f1, value, conf) {
    if ((value == 'normal') and (conf > 0.8)) {
        RateLimiterMO limit (f1.src, f1.dest, 0%);
    }
}

Horizontal Conflict

Fig. 2. Defining configurations for resilience is a multi-level problem, with vertical configuration conflicts across levels, and horizontal conflicts along the \( D^2 R^2 \) + DR strategy.

Fig. 3. (a) Goal decomposition of a resilience specification and refinement into low-level configuration policies, and (b) resilience strategy for the scenario (intermediary step).

which policies should be executed. Whilst [15] presents the theoretical foundations for this type of analysis, it explicitly leaves domain-specific information outside the scope of study.

However, domain-specific information is arguably necessary to detect conflicts. It is our intention to build on these general techniques for policy analysis and include domain-specific expertise. For example, consider a scenario with two concurrent challenges – a DDoS attack targeted at a server farm and a flash crowd event directed towards another resource. Because of the DDoS attack, some remediation policies may lead to rate limiting being invoked on access routers; a network-level mechanism. However, a different set of policies may indicate that in the presence of a flash crowd event, we could reasonably decide to replicate a virtual service to another server farm; a service-level mechanism. This is indicated as a vertical conflict in Fig. 2. In this particular example, because of the naïve rate limiting that is operating as a consequence of the DDoS attack, trying to replicate a service can make the resource starvation situation worse. Also, it is necessary to guarantee the correct enforcement of the interaction during run-time in which, for example, a device may fail and the interaction may have to be re-checked. Thus we intend to investigate how management patterns can be pre-verified for conflicts, possibly extending existing solutions and tools for policy analysis [16], [17].

C. Pattern and Policy Refinement

A given management pattern and its policies are expected to realise a high-level requirement to ensure network resilience, e.g., defined in terms of the availability of a server farm and the services that it provides. Whilst management patterns provide the means for establishing policy-based interactions, resilience strategies still have to be pre-specified manually. Although for a small case study it is relatively straightforward to specify the policies, more complex scenarios, for instance, that require cooperation across different autonomous domains, would make creating concrete policies by hand difficult. For this reason, toolsets are required to aid this process.

We will thus seek to (semi-) automatically derive implementable pattern and policy configurations from high-level specifications and requirements, with minimum human involvement. Considerable research has been directed to address the problem of policy and goal refinement [18]. One of the reasons for the lack of success so far is the difficulty of defining general-purpose goal refinement techniques. In contrast, by relying on application-specific domain knowledge
we intend to provide support for the generation of management patterns from higher-level specifications, through the use of goal refinement and planning techniques. For example, by decomposing specific sub-goals that independently realise the different phases of the $D^2R^2 + DR$ strategy, each phase can thus be refined into more concrete sub-goals, until it consists only of concrete implementable operations (Fig. 3). Furthermore, this also includes the possibility of refining pre-existing patterns over time, as a better understanding of the challenges affecting the network is developed, as part of the offline control-loop in the $D^2R^2 + DR$ strategy.

V. CONCLUDING REMARKS

We are developing an infrastructure for the management of networked mechanisms supporting a range of resilience strategies. Management policies provide the means for adapting the operation of these mechanisms in response to, e.g., abnormal network traffic, failures of components or performance degradation. In order to support the systematic design and evaluation of resilience strategies and the capture of best practices and the experience of network operators, we are investigating the development of a catalogue of reusable attack- and challenge-specific management patterns that can address a range of challenges, e.g., DDoS attacks, worm propagations, and so on. This requires an understanding of the effects of challenges and the optimal configuration of resilience mechanisms that can be used, which will then be weighed against the requirements of expected challenges.

ACKNOWLEDGEMENTS

This research is supported by the EPSRC funded India-UK Advanced Technology Centre in Next Generation Networking and has been partially supported by the European Union Research Framework Programme 7 via the PRECYSE project with contract number FP7-SEC-2012-1-285181.

REFERENCES


Dynamic Monitoring of Composed Services

Muhammad Asim, Bo Zhou, David Llewellyn-Jones, Qi Shi, Madjid Merabti
School of Computing & Mathematical Sciences
Liverpool John Moores University
Byrom Street, Liverpool L3 3AF, United Kingdom
{M.Asim, B.Zhou, D.Llewellyn-Jones, Q.Shi, M.Merabti}@ljmu.ac.uk

Abstract: Service-Oriented Architectures (SOAs) are becoming a dominant paradigm for the integration of heterogeneous systems. However, SOA-based applications are highly dynamic and liable to change significantly at runtime. This justifies the need for monitoring composed services throughout the lifetime of service execution. In this paper we present a novel approach for monitoring services at runtime and to ensure that services behave in compliance with a pre-defined security policy. Services are defined using BPMN (Business Process Modelling Notation) processes which can then be monitored during execution.

I. INTRODUCTION

Modern software architectures are increasingly dynamic in nature. Among them, Service-Oriented Architectures are one of the most prominent paradigms. SOAs allow software components from different providers to be exported as services for external use. Service descriptions (including both functional and non-functional properties) are published by service providers and are used by the potential users to discover services. A service composer is a service provider that is responsible for constructing service compositions and offering them to consumers. Service discovery is based on matching user requirements and security needs with the published service descriptions. Typically, service composers will have different needs and different requirements. They have varying business goals and different expectations from a service, for example in terms of functionality, quality of service and security needs. Given this, it’s important to ensure that a service should deliver what it promises and should match the user’s expectations. If it fails, the system should take appropriate subsequent actions, e.g. notifications to the service invoker or service composer. However, SOA-based applications are highly dynamic and liable to change heavily at runtime. These applications are made out of services that are deployed and run independently, and may change unpredictably after deployment. Thus, changes may occur to services after deployment and at runtime, which may lead to a situation where services fail to deliver what has been promised. Traditional validation and verification techniques cannot foresee all of these changes as they are mainly pre-deployment activities. Therefore, there is a need to shift towards runtime monitoring of services [1].

Aniketos project is an EU research project [2] that addresses trustworthy and secure service compositions with run-time monitoring and adaptation of services. The adaptation is needed due to changing operational, business or threat environments or due to changes in service quality and behaviour. Among the challenges is the monitoring of services at runtime to ensure that services behave as promised. This paper focuses on our proposed novel monitoring framework that is based on the runtime monitoring of a service to ensure that the service behaves in compliance with a pre-defined security policy. We mainly concentrate on monitoring service behaviour throughout the service execution lifetime to ensure that services behave in a compliant manner. The monitoring mechanism is based on the Activiti platform [3] and services are defined as BPMN [4] processes.

BPMN is widely used as a modelling notation for business processes. Before BPMN 2.0, analysts or developers used to receive a BPMN 1.x model for requirements or documentation, which had to then be converted into an execution language such as the Business Process Execution Language for Web Services (BPEL4WS, also known as BPEL) [9] for implantation and deployment. This could result in ambiguities and unexpected results. By contrast BPMN 2.0 provides a standard for both modelling business processes and implementing a process execution model. Both business oriented people and developers can speak with the same vocabulary and share business models without the need for conversion.

The rest of the paper is organised as follows. The next section presents an analysis of existing techniques. A monitoring approach is proposed in Section 3. Section 4 concludes the paper and indicates the direction of our future work.

II. RELATED WORK

As part of the work undertaken for the Aniketos project, we carried out a study on existing techniques relevant to the runtime monitoring of Service-Oriented systems. The result reveals that research in this area is still in its infancy and mainly conducted using XML-based service composition languages [5,7,8]. Existing methods do not consider the graphical representation of the business processes. This could introduce a semantic gap between the business processes are described and the way they are implemented as a service composition.

The work presented by Baresi et al. [5, 6] is based on how to
monitor dynamic service compositions with respect to contracts expressed via assertions on services. Dynamic service compositions are presented as BPEL processes which can be monitored at runtime to check whether individual services comply with their contracts. Assertions are specified with a special-purpose specification language called WSCoL (Web Service Constraint Language), for specifying constraints (monitoring rules) on service execution. The monitoring rules are then deployed with the process through a weaving procedure. The weaving introduces a proxy service, called a monitoring manager; which is responsible for evaluating monitoring rules. If some constraints are not met, the monitoring manager will inform the BPEL process about the enforcement. In Haiteng et al. [7], the authors propose a solution to the problem of monitoring web service instances implemented in BPEL. The solution uses a monitoring broker to access Web Service runtime state information and calculate the QoS (Quality of service) property values. The monitoring broker is devised with the support of Aspect-oriented Programming (AOP) that separates the business logic of the Web Service from its monitoring functionality. In Moser et al. [8], an event-based monitoring approach for service composition infrastructure is proposed. The framework intercepts each message as an event of a particular type and leverages complex event processing technology to define and detect situations of interest.

III. MONITORING FRAMEWORK

The idea behind the monitoring approach presented in this paper comes from Execution Listeners that can be configured in a BPMN 2.0 XML file. Activiti supports an extension on top of the BPMN 2.0 specification referred to as Execution Listener. Execution Listeners can be used on the process itself, activities and transitions. This provides a hook into the process execution that can be used for process monitoring. Execution Listeners allow executing external Java code or evaluating an expression when certain events occur during process execution. The following process definition contains three execution listeners, which can be seen highlighted in bold [3]:

```xml
//Sample code
<process id="executionListenersProcess">
  <extensionElements>
    <activiti:executionListener
      class="org.activiti.examples.bpmn.example.ExecutionListenerExampleOne" event="start" />
  </extensionElements>
  <startEvent id="theStart" />
  <sequenceFlow id="theStart" targetRef="firstTask" />
  <userTask id="firstTask" />
  <sequenceFlow id="firstTask" targetRef="secondTask" />
  <extensionElements>
    <activiti:executionListener
      class="org.activiti.examples.bpmn.example.ExecutionListenerExampleTwo" />
  </extensionElements>
</process>
```

The first Execution Listener is notified when the process starts. The listener is an external Java-class (like ExampleExecutionListenerOne, shown below).

```java
public class ExampleExecutionListenerOne implements ExecutionListener {
  public void notify(ExecutionListenerExecution execution) throws Exception {
    execution.setVariable("variableSetInExecutionListener", "firstValue");
    execution.setVariable("eventReceived", execution.getEventName());
  }
}
```

The general architecture of the monitoring framework that we use to monitor the BPMN processes introduced in this paper is shown in Figure 1. The Activiti engine with the help of Execution Listeners throws events for the deployed BPMN process. The framework consists of an Analyzer that accepts a set of security requirements (a monitoring policy) for a particular process to be monitored. The monitoring policy is defined by the service composer. The Analyzer then recovers the monitoring patterns that are related to the requirements from the Monitoring Pattern Repository and checks whether the received events are consistent with the patterns and not reports a violation through the Notification module [2]. The Notification module is a part of the Aniketos platform that provides a notification mechanism. The monitoring policy will be defined using the language ConSpec [10]. However, we have not integrated the language ConSpec into our monitoring framework at this stage. The components of the monitoring framework are shown in Figure 1. The details of the monitoring unit are as follows.
Event Manager: This module gathers the events coming from the Activiti engine and passes them to the Analyzer. In table 1 an overview is provided of the event types that can be configured in a BPMN 2.0 XML definition using the Activiti execution and Task listener extensions.

<table>
<thead>
<tr>
<th>BPMN construct</th>
<th>Event type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Start and end</td>
<td>A start and end event of a process instance</td>
</tr>
<tr>
<td>Activity</td>
<td>Start and end</td>
<td>A start and end execution listener of an activity</td>
</tr>
<tr>
<td>Transition</td>
<td>Take</td>
<td>A transition execution listener can catch a take transition event</td>
</tr>
<tr>
<td>User task</td>
<td>Create, assignment &amp; complete</td>
<td>A user task throws events when the task assignment has been performed</td>
</tr>
</tbody>
</table>

Table 1: An overview of event types that can be configured in BPMN using Listeners.

Monitoring policy: A set of requirements that specify what to monitor for a particular BPMN process. We again suggest the language ConSpec to specify the monitoring policy. A policy written in ConSpec is easily understandable by humans and the simplicity of the language allows a comparatively simple semantics. This will enable the service composer to easily specify the monitoring requirements for their processes and monitor them using this framework.

Consider the following example where a service composer creates a travel booking composition that consist of several tasks such as ordering, booking a hotel, booking a flight, payment and invoice, and each task is performed by a component service. The service composer of the travel booking composition might want the payment service component to only be invoked when it has a trustworthiness value greater than 90%. This requirement could easily be specified using the language Conspec as shown below.

```
SCOPE method
SECURITY STATE
Int trust_threshold=90%; /* assume trustworthiness is in [0%,..., 100%]*/
BEFORE invoke (service_A, args)
PERFORM
{eval_Trustworthiness(service_A) >= trust_threshold) -> skip
```

Analyzer: Upon receiving events from the Event Manager, it analyses them by accessing patterns from the repository. It uses the monitoring policy to select the appropriate monitoring patterns for a particular process.

Interface: This is used to specify definitions for monitoring patterns using monitoring IDs. These definitions are then stored in the repository. Drools Guvnor [13] is a centralized repository for Drools knowledge bases. It has a rich Web-based GUI, editor, and tools to aid in the management of a large number of rules. The repository gives the flexibility for storing
versions of rules, models, functions, and processes etc.

<table>
<thead>
<tr>
<th>amount min</th>
<th>amount max</th>
<th>period</th>
<th>deposit max</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low-Application</td>
<td>[application]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>amount [€]</th>
<th>amount [€]</th>
<th>lengthYears [4]</th>
<th>deposit [€]</th>
<th>type [€]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Drools Guvnor graphical editor.

Notification Module: This is mainly used by the Analyzer to report any violations. The Notification module is based on publish/subscribe paradigm. In such case, the notification module can forward notification to relevant subscribers according to the subscription criteria. The subscriber can be relevant services or end-users.

We intend to implement these components as part of the Aniketos projects in order to provide a comprehensive and flexible service monitoring framework.

IV. CONCLUSION AND FUTURE WORK

In this paper, we presented a framework for monitoring the compliance of BPMN processes at runtime using monitoring patterns. The monitoring framework is based on the Activiti platform to deploy BPMN services and generate events throughout the service execution lifetime. The events will be monitored to ensure that a service behaves in compliance with a pre-defined monitoring policy. We suggested using the language ConSpec for specifying the monitoring policy. ConSpec is a simple and easily understandable language for specifying service contracts. The framework will help organizations to automate the monitoring of business processes and proactively report compliance violations.

Our future work includes the implementation of our proposed framework for some real life scenarios. We also hope to integrate and implement the language ConSpec in our framework for specifying the monitoring policy.

REFERENCES


Susan Appleby, CESG
Chris P, CESG
Colin Carter, Aurora Consulting

Introduction

The discipline of designing a security architecture is relatively immature and is generally lacking in foundational principles. Although there is a large and growing field of expert practitioners, they vary in approach and there is a tendency to over focus on security products at the expense of examining security aspects of the overall system. In practice there are many factors which contribute to the overall security posture of a system, but these are not generally documented or modelled explicitly.

CESG is the National Technical Authority for UK Government Information Assurance. We produce policy and guidance for UK Government on how to create secure information systems. As part of this role, we review numerous systems and networks for a wide range of organisations from central government to small private companies. This work has given us a broad appreciation of how different groups approach designing networks and the challenges facing security reviews of those networks. As there is no standard way of presenting the security aspects of the design, it is often difficult to extract the necessary information from the overall system design. In particular, it is often unclear what security controls have been put in place and how they fit into an overall security strategy to protect sensitive information.

Security Documentation & Formal Modelling

This presentation presents the findings of research and workshops into what information is required for a security architect to be able to judge the security of a given network, and how to represent this diagrammatically.

Identifying the key pieces of information required allows for “quick wins” in terms of setting standards for the documentation of systems being reviewed, however it still leaves an undesirable level of subjectivity in the process. We therefore built on the work performed in identifying the key pieces of information required and used this as a basis for a formal model.

The model represents networks from the lowest level, i.e. physical cables, through to very high level constructs such as security domains and security boundaries. This allows for a network to be accurately represented from an enterprise architecture point of view as well as a security point of view.

We will present the concepts in the model and our design rationale along with a discussion of how the work can be extended in the future.
Method

One of the key requirements of our work to create a model for network security was to ground it in the reality of what is actually useful in a business context.

We started this research work by analysing documentation sets provided to us by various different organisations for design reviews. We picked out the key pieces of information that we commonly required and highlighted what was commonly missing. This allowed us to start to gain an understanding of what would be required.

We applied the insight this gave us to a case study, based on a local government network, a moderately complex scenario involving multiple physical sites, several different levels of assets, and a number of different trusted partners. We created a series of different “Security Views” of the system, each representing different aspects of the security critical information for this particular system. The views were developed following an iterative process of designing diagrams and holding workshops with security professionals to test their validity and completeness.

Six security views were defined: Logical network; Logical security domains; Data flows; Switching network and virtualisation; Management network; and Site security.

This work was then used as a basis for deciding on the key elements required for a security meta model, and how they should link together.

Security Meta Model

Security views are a useful way of representing the security relevant aspects of network architectures. However, there is no underlying formalism yet, which leaves scope for different interpretations and designs. Neither does it allow for any automated visualisation of networks or automatic creation of several views of the same network. To address these issues, we need a model which can be used to represent arbitrary networks.

We focus here primarily on technical attacks and defences, and therefore adopt a system oriented model with capabilities for modelling human elements as opposed to a human oriented model with support for modelling systems. Extending the model to better represent the human elements is a longer term goal.

The model is extensive, and we can only give a flavour of it in this short paper.

To begin, we first consider one of the model’s fundamental concepts: a “System”. A System can be an individual device, such as a computer, or it can be a collection of devices. This was introduced for situations where there are a number of devices which are, to all intents and purposes, identical. For instance an office with 100 machines sharing a common build all sitting on the same floorplate could be represented as one instance of a system.
We describe connections between systems and model security functions in the Systems layer model. The key aspects of this layer are:

- How systems communicate (data flows)
- How systems reside in security domains and how data crosses the domain boundaries
- The technical defences that exist at domain boundaries and on systems
- How one or more security products form a technical defence

Taking security functions as an example, the model appears as follows (Figure 1).

In Figure 1, we can see that a System Security Function is implemented on a System (for example, a hardware firewall device will have some kind of firewalling software running on it). This System Security Function is configured by a set of Security Rules. A collection of Security Rules (possibly relating to multiple System Security Functions) form a Technical Defence.

The security function elements can be seen in the context of the full systems layer model in Figure 2.

Underpinning the Systems layer model are more detailed models of networking and virtualisation. In the networking part of the model, we represent enough core networking to accurately model connections from the physical to IP layer. This is important as to gain a true appreciation of the security of a system, everything down to the physical connections must be captured. The networking layer model is shown in Figure 3 below.
Future work

Although it is extensive, this model is not presented as a final solution – rather as an initial draft. There are placeholders in the model for strengthening the following aspects, for example:

- Physical security
- Attack modelling
- Risk modelling
- Human elements

There is also significant potential for improving the visualisation aspects – particularly of the Security Views.

The key development we feel is required, however, is that of tooling. Tooling could allow architects to utilise this model during the design and development of new systems. Views and diagrams of security critical information could then be automatically generated throughout the lifecycle, making it far easier for security professionals to review and assess the system. We feel that the complexity of the model could be mitigated by a fair degree of automation.

Looking further ahead, it may be possible to automatically analyse the design and highlight high risk components, or to suggest design patterns for common elements.
Towards a Simulation of Information Security Behaviour in Organisations

Martin Ruskov, M. Angela Sasse
Information Security Research Group
University College London
London, UK
m.ruskov(a.sasse)@cs.ucl.ac.uk

Paul Ekblom
Design Against Crime Research Centre
Central Saint Martins
London, UK
p.ekblom@csm.arts.ac.uk

Abstract—In this paper we propose the fundamentals of a design of an exploratory simulation of security management in a corporate environment. The model brings together theory and research findings on causes of information security risks in order to analyse diverse roles interacting through scripts. The framework is an adaptation of theoretical and empirical research in general crime prevention for the purposes of cybercrime. Its aim is to provide insights into the prerequisites for a more functional model.

Keywords—information security; conjunction of criminal opportunity; crime scripts; simulation

I. INTRODUCTION

Research in the recent decade, e.g. [1], [2] demonstrates the importance of addressing the human factor in information security: attackers often obtain information, access to systems or money by tricking customers or employees. This calls for security models that capture the complexity of the wider socio-technical system. Examples of such models are the Mechanics of Trust [3] and the Compliance Budget [4]. Of interest is the unification of these models in the search for a more general theory of how we can design systems that prevent this sort of attack.

In 2004 in a report for the UK Government Foresight Programme, Collins and Mansell [5] suggest the adoption of a framework from conventional crime science - the Conjunction of Criminal Opportunity (CCO) framework [6] - as a basis for developing and designing resilient systems and effective cyber defences. The CCO framework presents a systematic and conceptually rigorous categorization of immediate contributing causes to criminal events (with the potential to trace back to distal causes). Compared with prior frameworks it captures a much wider range of causes for attacks. The CCO framework unifies traditional offender-oriented with situational crime prevention and is explained further in this paper along with its potential application in cybercrime. It has already been suggested [6] that CCO can combine well with the Crime Scripts approach [7] in which the various agents interact with one another and exploit or harm the technical settings, often combining factors from the physical environment and cyberspace.

This makes CCO generally applicable, yet simple enough to serve as a framework in which other theories of causation of criminal events could be unified in an exploratory simulation. Such a reusable framework that incorporates domain knowledge could be seen to be very similar to the formal patterns used in software engineering and architecture [8]. A potential resulting simulation can deliver a learning experience exploring an actual research framework. Moreover, challenges encountered during simulation design provide feedback that tests the consistency of the theoretical research, as has been done with simulations in conventional crime science by Birks, Townsley and Stewart [9].

Quantitative simulation models already exist. For example, the Naval Postgraduate School developed a game to spread awareness about cyber security called CyberCIEGE [10]. It is a customizable platform that allows designers to develop scenarios for their organisations. A typical scenario in CyberCIEGE is about preventing users from letting malware into the corporate intranet, preventing social engineering and safeguarding data. Unlike CyberCIEGE however, the simulation model proposed here seeks theoretical (and eventually empirical) validity.

II. BACKGROUND

In his research on insider attacks Schultz [11] considers the CMO model consisting of capability to commit attack, motive to do so and opportunity to do it. In his work Schultz also reviews a model insider attackers by Tuglular and Spafford [12] allegedly featuring factors describing such as personal characteristics, motivation, knowledge, abilities, rights and obligations, authority and responsibility within the organisation, and factors related to group support. Parker [13] develops the SCRAM model. The abbreviation reflects the factors considered: skills, knowledge, resources, authority and motives.

The CCO framework considers eleven overarching classes of complementary causes which come together to make a criminal opportunity. The number of these classes may seem high, but that is to model all potential contributory causes of crime in the real world. To model the contributory causes of cyber crime, the classes of causes can be considered to fit in three wider groups - personal, technical and social factors. These all are represented in the figure.

The personal (attacker) factors are:
- **Criminality** and wider personal traits influencing attacker’s tolerance towards immoral or criminal deeds. This is where personal characteristics [12] are being addressed.
- **Anticipation of risk, effort and reward** - the rational decision and utility behind the attack
- **Abundance of resources to commit crime** - both cognitive resources and capabilities, and social factors such as trust, but also technical hacking tools - the attacker needs to be both aware of their existence and be able to operate them. Schultz’s [11] capability and opportunity can be viewed as
part of the cognitive resources of inside attackers. Parker’s [13] resources fall naturally into this category, but also his skills, knowledge and to some extent authority.

- Immediate **readiness to undertake an attack**, e.g. the commonly modelled motives to do it like disquietment [11], [13].

- **Lack of skills to avoid crime** - potential skills that would reduce attacker’s need to commit crime. For insiders this could be ability to manage stress, soft skills to improve common understanding of potentially discouraging issues, etc.

- **Attacker presence in situation** - circumstances like the fact that this person is part of the organisation, but also that they have certain access privileges that might allow them to abuse the organization.

- The **wider environment** which could contribute towards attacks and discourage or restrict potential preventers. Authority can be thought of as a feature of the organisational environment that allows misuse, e.g. lifting doubt from people with authority that demand information which they don’t possess.

The issue of authority as considered by previous models comes to illustrate the interactions of these different factors. On one hand authority is a resource that a potential offender possesses and enables them to commit the attack. On the other one’s authority also is a function of the wider environment - it is the organisational culture that allows authority to be exercised without protective questioning from others.

**III. Crime Scripts**

One technique used in conventional crime prevention is that of crime scripts [7]. These are sequences of events characteristic of recurring, or similar, attacks. Typically a script would also capture the preparatory and consummatory sequences of actions that attackers engage in before and after the actual attack, thus providing a wider picture and context to how attacks commonly happen.

Research in design against crime has revealed that not only are the scripts of attackers insightful tools, but also the scripts typical of ordinary users who may or may not be acting in the role of crime preventers [6]. Being a normal business routine, this category of non-criminal scripts can possibly be extracted from the established business processes in an organisation. These processes could either be designed or ad-hoc; with the former it could be that the actual routine is different from the one designed, e.g. in cases when users find it difficult to comply with designed procedures [4].

Sample of scripts of attackers and regular users are illustrated below:

Users’ generic script:
1. User is asked to create an account.
2. User generates a password, according to own cognitive scheme.
3. User stores password (either mentally or physically)
4. If stored physically password needs to be secured.
5. User devises restoration procedures, e.g. reminder question or location to where file is stored.
6. Generation of procedure to restore passwords.
7. Using password (or subset of its letters) for authentication.
8. Make sure password is not retrievable after end of use.

Script instance illustrating misuse:
1. Change password (due to a system request to change regularly).
2. Store new password on a piece of paper, convenient for frequent referencing.
3. Because it is already stored externally and thus easier to reuse, use new password for several systems in company.
4. Skip anti-virus checks so that they do not slow down work.
5. Reschedule anti-virus checks for non-working hours.
6. Log out when done working for the day.

Hostile insider script:
1. Decide to cause harm to company and identify target and what to do with it (e.g. upload internal financial data onto internet).
2. Choose mode of access and tactics to reach target (e.g. plan on time when target company’s office will be unlocked and empty).
3. Acquire target’s externally stored password (e.g. on piece of paper).
4. Access target system, retrieve asset and copy it (e.g. physically access office computer of Accounts department, deploy trojan).
5. Distribute valuable asset to market analysis.

External attacker script:
1. Try target technical system for potential backdoors or exploits.
2. Research social networks of potential human targets that have access to system.
3. Infect a computer of trusted peer of user.
4. Have trusted computer send trojan to user.
5. Infect computer with access to secured network.
6. Copy data out of the secured network.

IV. SCRIPT CLASHES

Scripts on their own depict the procedural nature of everyday and criminal situations. However, they represent the routine of an individual and not the dynamics resulting from the inevitable interaction of these routines upon an encounter between their corresponding performers. For example, if a potential attack is being suspected, security officers could decide to reduce the number of people that have access to valuable data. This would be a counter-move trying to prevent data theft. It is natural that when seeing the new obstacle the attacker would decide to change their tactics of access. As a result devising a counter-counter-move which could be trying to get access through a unsuspecting colleague, trusted enough to still have the wished access.

Such a process of interruption and branching from routine scripts demonstrates the complexity of dynamic modelling of crime, and cybercrime in particular. Over time, the scripts and counter-scripts become steadily more elaborate. But there may be only a limited number of archetypical script clashes to address (such as ‘conceal move versus detect move’) [6].

V. CONCLUSIONS AND FUTURE WORK

In the process towards a simulation of cyber crime this position paper contributes in two ways. First, it proposes an adaptation of the CCO framework to information security. And second, it suggests modelling of script clashes and discussion of potential reactive adaptations of scripts. These two patterns of representation both contribute to the fundamentals of a design for a simulation of information security behaviour in a potential crime situation.

Designing and implementing a simulation of information security behaviour is a challenging task. A computer simulation typically requires a finite (and thus mathematically closed) representation. On the other hand there is an arms race between attackers and security officers which requires continuous adaptation and innovation [14] - a potentially infinite space of ideas or steps. A way out of this contradiction could be to address only the recurring attacks, but not the innovative ones. This way the hope is to get coverage of the “20% of scripts occurring 80% of the time”.

The domain of simulation can be built utilizing the currently developed CCO browser game prototype [15]. This prototype features neither any simulation elements, nor crime scripts yet. Instead it guides users through a facilitated process of brainstorming crime interventions. Still, the game prototype could be used to analyse attacker scripts thus collecting user generated counter-moves. Data will be collected with the prototype until certain level of saturation of ideas is achieved. The collected data could then be used to describe (hopefully enumerate) the space of counter-moves within certain abstraction and simplification.

REFERENCES


Patterns of Information Security Postures for Socio-Technical Systems and Systems-of-Systems

Tim Storer, Karen Renaud
School of Computing Science
University of Glasgow
Glasgow, Scotland, G12 8QQ

William Bradley Glisson
Digital Forensics Laboratory
University of Glasgow
Glasgow, Scotland, G12 8QQ

Abstract—This paper describes a proposal to develop patterns of security postures for computer based socio-technical systems and systems-of-systems. Such systems typically span many organisational boundaries, integrating multiple computer systems, infrastructures and organisational processes. The paper describes the motivation for the proposed work, and our approach to the development, specification, integration and validation of security patterns for socio-technical and system-of-system scale systems.

I. INTRODUCTION

The concept of a software design pattern, a tailorable solution to a recurring design problem, was first popularised by Gamma et al. (1994). These patterns addressed small scale, object oriented, software design problems, typically involving the arrangement of a small number of software classes. Design patterns have also been applied at other scales in software and systems engineering, including software architectures (Shaw and Garlan (1996) is one of many texts on the topic) and the analysis of patterns of work (Martin and Sommerville, 2004).

There has been considerable interest in the application of patterns to a range of information security domains. Security patterns have been proposed in contexts such as distributed access control (Erber et al., 2007), operating systems (Fernandez et al., 2006), and web application development (Kienzle and Elder, 2001). There have also been surveys of the state of the art, Yoshioka et al. (2008) for example, and a book on development with security patterns (Schumacher et al., 2005).

This existing research has addressed the application of a patterns approach to information security design problems at the software system scale. However, as Siponen and Heikka (2008) also noted, there is currently a lack of established engineering practice, tools or methods that address information security design problems at the socio-technical system and system-of-system scale. Such systems are based around information technology (IT) applications and infrastructures, and can be characterised as:

- very large scale, consisting of multiple heterogeneous organisations, work flows, actors, software applications, middlewares and hardware platforms; and
- very long lived, with evolutionary development occurring across organisational boundaries as the system context changes, users change, individual components are updated and new requirements are established.

The challenges of addressing general engineering problems for systems at these scales is well reported in the literature (Sommerville et al., 2011; Boardman and Sauser, 2006). Diverse examples of such systems include inter-agency civil emergency response (Sommerville et al., 2009a), logistics management (Baskerville and Siponen, 2002), election and voting management systems (Lock et al., 2008), space based infrastructures (Boin and Fischbacher-Smith, 2011) and electronic trading systems (Sommerville et al., 2011). The adoption of virtualised, remote and distributed cloud computing infrastructures will exacerbate the complexity of engineering at this scale.

It is recognised that such systems are vulnerable to information security threats that are both evolving and increasing in scale and complexity. Future ‘cyber’ attacks will be directed at multiple systems, infrastructures, organisations and nation states simultaneously (Research Councils UK, 2011; UK Office of Cyber Security and UK Cyber Security Operations Centre, 2009). Threats will combine both external activity and malicious ‘insider’ behaviour. In the future, systems that enable and support collaborative information security across organisational boundaries will be essential. To respond to this challenge, there is a need to develop engineering methods that address the design of security into systems at the socio-technical and system-of-systems scale.

To begin to address this challenge, this paper argues for the development of:

- methods for the identification, documentation and validation of design patterns of information security postures for socio-technical systems and systems of systems;
- an initial repository of validated design patterns which can be applied to recurring security problems at the socio-technical and system-of-systems scale;
- a method for selecting and tailoring a design pattern to a specific system context; and
- methods for evaluating the suitability of a proposed socio-technical security design for a specific context through modelling and simulation.

The work proposed will adapt the well established pattern approach to system design to the socio-technical and system-of-system scale. The pattern repository will provide a source of guidance for engineers seeking to address large scale...
information security design problems. The application method will allow design patterns to be selected and tailored to specific socio-technical contexts. This approach will allow existing craft and experience to be formalised in engineering methods and tools.

This paper describes the initial results of this work, and is structured as follows. Section II illustrates our approach to documenting information security design patterns for socio-technical systems. The patterns approach is illustrated using a detailed example of a security incident response team in an organisation. Section III briefly enumerates other patterns that we have identified, illustrating the breadth of applicability of the approach. Finally, Section IV summaries the work and points to future directions. In particular, we describe future work on the the validation of socio-technical system designs through the use of multi-agent simulation.

II. DESCRIPTION OF METHOD

Following the patterns approach, a Socio-Technical Information Security Design Pattern will include:

- **The context of an socio-technical security problem, explaining the circumstances in which the pattern can be applied.** The context section of the pattern describes the configuration of organisational vulnerabilities, known threats and attacker capabilities that can be addressed or mitigated through the application of the pattern. This part of the pattern is equivalent to a threat model that is typically described before a security design is proposed to address it.

- **A template for a socio-technical solution combining information systems, human actors and processes.** Specifically, the template solution will describe the roles and responsibilities of relevant actors within the organisation, and the communication channels and information requirements for the actors to discharge their responsibilities.

There are several notations for modelling socio-technical systems, notably: goal based notations, such as TROPOS (Giorgini et al., 2005); and the SysUML (Hause, 2006). There is also the STAMP method for identifying systemic hazards in systems of systems (Leveson, 2004). However, we have found the notion of responsibility useful in analysing and describing socio-technical systems in terms of the actors in an organisation, the responsibilities they hold and discharge and the information they require and produce (Sommerville et al., 2009b).

Lock and Sommerville (2010) demonstrated the feasibility of using an extension of the responsibility notation to capture key aspects of systems-of-systems. Consequently, we anticipate that the graphical notation (Sommerville et al., 2009a) we have previously developed for modelling the socio-technical and system-of-system contexts will be appropriate for describing socio-technical security pattern templates, in a similar way to the specification of software design patterns in UML.

- **The trade-offs, including any disadvantages, that must be considered when applying the pattern.** This part of the pattern will be developed through analysis of previous case studies and work with industrial partners. We will identify both the benefits of applying the pattern for information security at the socio-technical level, as well as any risks that are introduced.

- **Any related socio-technical and/or software security patterns.** Many design pattern schemes list relations within the same family of patterns, so that, for example, patterns which complement each other can be identified. In this section, we will identify related socio-technical information security patterns that complement one another. In addition, we will identify software security patterns, already documented in the existing literature, that complement a socio-technical pattern. This approach will support an integrated socio-technical approach to system security, allowing the consequences of security design decisions at different levels of abstraction and scale to be evaluated.

We briefly sketch an example socio-technical security pattern to illustrate our proposed approach. Figure 1 illustrates the pattern for a Security Incident Response Team (SIRT).

The context describes the threat model in which the pattern is applicable: when an organisation perceives some information security breaches as inevitable due to a large, distributed, heterogeneous infrastructure or rapidly changing insider capabilities, which make a secure system difficult to achieve. In addition, the pattern is applicable when it may be cheaper to respond to and mitigate attacks, rather than attempting to prevent them occurring at all.

The solution template responsibility model contains five actor roles, denoted using the UML stick figure stereotype. The Incident coordinator, responsible for collecting incident reports and initiating investigations. Employee represents a general member of the organisation. The Automated monitor is a software component that can be configured to report suspicious system activity, such as inappropriate file accesses, failed login attempts or firewall probes. The Incident manager, responsible for directing the focus of a specific investigation. The Incident investigator, responsible for gathering and analysing evidence under the direction of the investigation manager.

Note that the model describes actor roles, rather than specific actors. Consequently, the pattern is applicable to both large organisations, where the role of incident coordinator, manager and investigator may be distinct, and smaller organisations, where all these roles may be undertaken by a single IT worker.

In addition, actor roles may be undertaken by humans, organisations or technical components. In the responsibility modelling notation, any object can be modelled as an actor if the modeller perceives the actors to have intent and be able to hold a responsibility. This flexibility makes responsibility modelling useful for capturing different scales of socio-technical system.
### Title: Security Incident Response Team (SIRT)

#### Problem Context

An organisation maintains a diverse collection of interconnected systems and infrastructures, used by many different actors with rapidly changing privileges. Attackers may be either external, or internal, exploiting one or more vulnerabilities in the system.

It is not considered feasible to prevent every security incident, but the extent of damage caused by attacks on the systems is often dependent on the time taken to respond to the attack.

#### Solution Template

<table>
<thead>
<tr>
<th>Employee</th>
<th>ReportingProcedure</th>
<th>Publicise reporting procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Monitor</td>
<td>Report suspected incident</td>
<td>Collect incident reports</td>
</tr>
<tr>
<td>Incident Coordinator</td>
<td>Initiate investigation</td>
<td>Coordinate investigation</td>
</tr>
<tr>
<td>Incident Manager</td>
<td>Collate and report findings</td>
<td>Escalate incident</td>
</tr>
<tr>
<td>Investigator</td>
<td>EvidenceReport</td>
<td>Collect and analyse evidence</td>
</tr>
</tbody>
</table>

#### Advantages and disadvantages

The pattern provides the organisation with a means of identifying and responding rapidly to evolving security incidents. There is a risk that the system will generate too many false positives (particularly in the aftermath of a breach) swamping the organisation’s ability to respond effectively.

#### Related patterns

Security policy management team; Secure logging framework; Incident reporting information system

---

Fig. 1. An example socio-technical security pattern.
The model illustrates the responsibilities held by each of the roles, and the information that is required or produced by them. Responsibilities are denoted in lozenges, while information resources are represented as UML classes (it is possible to specify class attributes if desired). An agent holding a responsibility is denoted using a UML association dependency. The UML transition dependency is used to denote the production or requirement of information to discharge a responsibility.

The incident reporting procedure and the responsibility of employees to report incidents. The model shows that an employee is responsible for reporting suspicious events for further investigation. The Incident Coordinator will undertake an initial assessment and initiate a further investigation if deemed necessary. Part of this responsibility is to appoint a manager for the incident, responsible for coordinating investigation efforts. The incident manager will direct the evidence gathering efforts of investigators and collate them into a full incident report. Finally, the Incident Coordinator is responsible for deciding whether to escalate an incident or not.

The pattern explains both the advantages and disadvantages of the SIRT model. In particular, it is noted that the system may generate too many false positive reports, leading to resources being consumed investigating non-incidents. This may be a particular problem where poorly trained employees are required to report incidents, or where automated monitoring software is not correctly configured. There may also be variability in incident reports, particularly in the aftermath of a reported breach.

As can be seen from the figure, socio-technical security patterns modelled on responsibilities can also be used to denote the information that is required and produced through the discharge of a responsibility. This documentation can then be used to identify the requirements for the supporting technical and administrative of an organisation. These requirements can sometimes be expressed as related design patterns (both technical and socio-technical).

In the pattern described in Figure 1, for example:

- The Employee and the Automated Monitor needs a means of communicating an incident to the Incident Coordinator in a standardised way. A standard layered information system for collecting reports should support this requirement by providing a software interface to the automated monitor and a user interface to the employee.
- The incident manager and investigators need information concerning the state of parts of the technical infrastructure in the lead up to an incident. The socio-technical security pattern thus drives the need for a secure logging framework for the technical infrastructure.
- There needs to be a means of ensuring that employees of the organisation are familiar with their responsibilities to report incidents and the procedure for doing so. Consequently, the security policy management team pattern is recorded as a related pattern. This pattern describes how the organisation manages and communicates security policies to its employees.

The example illustrates how socio-technical security patterns can provide an over-arching and integrating framework for software level security patterns.

### III. Other Patterns

This section of the paper provides a brief catalogue of further (elaborated) socio-technical security design patterns that we have identified at the time of writing:

- Security policy management team, including a security policy coordinator for the organisation.
- Inter–organisational response team, providing a means of communicating security incidents between organisations within a trusted environment.
- Inter-organisational ad-hoc secure channel, inspired by the security breaches (and remedies) encountered during data transmission between the United Kingdom’s Revenue and Custom and National Audit Office (Poynter, 2008). The pattern provides a means for secure communication when relationships between agencies are ad-hoc and intermittent.
- Credential and authorisation manager, provides a single point of contact within an organisation for registering new users, removing privileges of departing employees and managing authorisations as roles change.
- Software patch management team, providing for a disciplined approach to evolving software configurations in a large-scale heterogeneous Information Infrastructure.
- Off-site fail-over, prevents the organisation from failing should the main IT infrastructure site be damaged due to man-made or natural disaster.
- Intra–organisational firewall, allows an organisation to partition information internally to prevent conflicts of interest arising.
- Secure decommissioning process for data storage devices, ensuring they are disposed of without compromising organisational data security.

Space considerations prevent a detailed description of these patterns. However, the examples listed (and others) hint at the breadth of the applicability of the approach.

### IV. Summary and Future Work

The global information rich society is increasingly dependent on large scale computer based systems and infrastructures. These systems are increasingly threatened by threats capable of mounting attacks of increasing sophistication and scale. We believe that the development of patterns of socio-technical security postures addresses these security threats at an equivalent scale. This paper has outlined our envisioned approach, including the use of responsibility modelling as a core abstraction.

A key challenge at this scale is the validation of designs prior to their deployment. Mistakes in system design can be much more expensive to correct once implementation has begun. In the current state of the art, validation of the design of large scale, socio-technical systems is notoriously difficult. A trial and error approach, until an acceptable design is achieved,
is often employed. This can often result in system failures during initial deployments as problems are gradually rectified. Alternatively, systems may fail catastrophically and projects abandoned if socio-technical failures cause rapid declines in confidence in a system.

We propose to use the development of executable simulations of large scale systems as a means of validating socio-technical security patterns. We are investigating the use of multi-agent technologies that incorporate the responsibility modelling technique for capturing socio-technical structures. A repository of patterns will provide a re-usable collection of designs that can be applied to a multi-agent simulation of a socio-technical system, prior to deployment. Our vision is that the validity of applying a given socio-technical security pattern to a design problem can be tested first through simulation before a decision to commit to a substantial re-organisation is made. This paper has described the first steps in this direction.

REFERENCES


A Strategy for Formalizing Attack Patterns
Clive Blackwell
Department of Computer Science
Oxford Brookes University
OXFORD OX3 3JH. UK
CBlackwell@brookes.ac.uk

ABSTRACT
We have created a framework for modeling security that divides computer incidents into their stages of access, use and effect. In addition, we have developed a three-layer architectural model to examine incidents with the social, logical and physical levels. Our ontology that combines the architectural and incident models provides the basis for a suitable semantics for attack patterns, where the entities and relationships between them are accurately defined and understood. The current informality of these patterns means that their utility is limited to manual use, so we plan to extend existing work on formalizing design patterns to attack patterns, to aid the automated creation of effective defensive controls. A specification in logic, which is progressively refined into code, is a common method of developing high integrity and secure software, but there are additional issues in system protection, as there are several defensive controls rather than a single program. The attack patterns form a logical specification, which is intersected with the model of the defense to determine the corresponding defensive patterns to counter the attacks. This would allow better reasoning about possible defensive response measures, and holds out the possibility of proving security against certain attacks. We outline a roadmap for formulating attack patterns in our ontology and then translating them in logic, and will go into more detail in following work.

Keywords
Attack pattern, CAPEC, defense pattern, incident questions, logic

1 INTRODUCTION
We have created a framework for modeling both security [1] and forensics [2] that divides computer incidents into their various stages of access, use and effect. This enables us to consider prevention, detection and response at different times and locations to provide coherent and comprehensive defense-in-depth. In addition, we have a simplified three-layer architectural model to analyze incidents, which enables us to consider comprehensive attack surfaces [3] at all levels to defend against all modes of attack, whether logical, physical or social. The three aspects of vertical level, spatial scope and temporal progression are fundamental aspects of our security ontology [1] to enable modeling incidents in their entirety. We also indicate how our ontology can be represented in logic, which will enable the translation of attack patterns into logic.

Evidently, our architectural three-layer model shown in figure 1 is inspired by the OSI model, but we explicitly model people at the social level rather than their logical user accounts, and the physical world that is underneath the purely logical OSI network model. In addition, our logical layers are also more abstract and general, as they incorporate all computational aspects, including storage, processing and control, not only networking. More details on these and other design issues such as the sublevels are described elsewhere [4].

![Figure 1 – The multilevel architectural model](image)

We focus on the logical level concerning attack patterns, but the other two layers are also important. The social layer at the top includes people and organizations along with their goals and intentions, which are necessary in understanding attacks in their totality, and have their own sections within attack patterns.

The logical layer in the middle contains computers, networks, software and data, and has five sublevels to help with detailed analysis. Each level has its own semantics and different conception of location and time, which is important to consider, as attacks must be recognized, understood and reacted to by defensive controls with differing observational and control abilities. We note that any logical event ultimately has a physical existence that may be used indirectly to recognize attacks, and could therefore be part of a defense pattern, but we do not discuss the physical level further [2].

Howard and Longstaff invented a classification for network security incidents [5,6] that shows the different types of entity involved in attacks, their characteristics and the relationships between them. We extended the model, inter alia, to divide attacks into their individual stages within our incident classification scheme, as each stage has a particular purpose in supporting the incident goals, possibly performed by different stage actors with differing motivations and abilities. The three main active stages in an incident are system access, target use and incident outcome, along with the incident prologue and epilogue stages to make five stages in total.

In an active incident stage, the actor uses a method to perform an action that executes a threat to exploit a vulnerability with an immediate effect on a target. The incident has a social-level goal that is met by the ultimate effect, whereas the immediate stage
effect is only a means to an end and may have no direct impact. This recognizes that attacks involve multiple actions that can possibly be recognized by the defense, which should be included within the attack patterns as the main attack step may be impossible to stop directly. This is one advantage of attack patterns over attack trees that only model the main incident steps and not the wider context.

The Zachman framework [7] is a complex model for designing enterprise computing architecture with a two-dimensional grid, where six questions are posed to describe the different aspects of the system, which are answered for each of five conceptual levels. These six questions are who, what, why, when, where and how.

We use these as incident questions within our incident framework to guide systematic analysis. We extend Zachman’s framework with two more questions: with what is the means of attack, and to what is the target. The eight questions are answered for the entire incident and each stage to help establish comprehensive incident analysis. Each question has a relationship with a stage entity as shown in table 1. These questions are used to structure the attack patterns, and are also fundamental in their translation into logic.

<table>
<thead>
<tr>
<th>Question</th>
<th>Incident entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who</td>
<td>Perpetrator</td>
</tr>
<tr>
<td>What</td>
<td>Effect</td>
</tr>
<tr>
<td>Why</td>
<td>Motivation</td>
</tr>
<tr>
<td>To what</td>
<td>Target</td>
</tr>
<tr>
<td>With what</td>
<td>Resources</td>
</tr>
<tr>
<td>Where</td>
<td>Path</td>
</tr>
<tr>
<td>When</td>
<td>Stage</td>
</tr>
</tbody>
</table>

2 ATTACK PATTERNS

Attack patterns [8] are based on design patterns, made famous in the software engineering community by the Design Patterns book [9], which were themselves originally based on Christopher Alexander’s architectural patterns for designing buildings [10]. A design pattern is a general reusable solution to a commonly occurring problem in software design, and therefore represents desirable, intentional and expected system behavior.

Attack patterns apparently appeared around 2001 in the community, originated in the literature by Moore et al [11], and later elaborated [12]. Subsequently, attack patterns were classified in the Common Attack Pattern Enumeration and Classification (CAPEC) schema [13]. Attack patterns have similar uses to attack trees [14], but they have wider scope in considering the complete attack context rather than the malicious actions alone, which our incident classification helps to capture.

An attack pattern is analogous to a design pattern, as it describes how a particular type of attack is performed as a conceptual pattern. However, attack patterns are specified from the attacker’s viewpoint and they therefore represent undesirable, unintentional and unexpected operational behavior. They usually include sections for countermeasures, although we consign these to a separate defensive pattern for clarity. Otherwise, there is not much difference to design patterns, with a few new fields to model additional concerns, and some existing fields amended to represent the attacker’s viewpoint.

We take most sections of our attack pattern template from Fernandez et al [15], and include some other fields from our existing work [16] and elsewhere [8]. We clearly distinguish between attack and defense, as the attacker’s activities may be difficult to find or attribute even if all attacks could be completely characterized, which they cannot. The information available to the defense may be inadequate, as the incident may not be observed at all or rapidly enough to act in time, or the attack may not be understood or discriminated from legitimate activities.

We describe how to trace and remediate attacks using corresponding defense patterns using proactive avoidance, timely detection or later response measures, which uses our incident progression model to analyze possible defenses at each stage. In addition, we use the semantics of entities at different levels and locations from our architectural three-layer model to help recognize the different types of information observable and possible response measures available. Therefore, we separate out the idea of attack pattern from possible observation and response measures, which are transferred to the corresponding defense pattern that contain the countermeasures and evidence sections.

<table>
<thead>
<tr>
<th>Name:</th>
<th>Generic name for attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classifier:</td>
<td>Each pattern should have a unique identifier, as the same pattern may be known by different names</td>
</tr>
<tr>
<td>Perpetrator (who):</td>
<td>We include the general class of perpetrator as an addition to Fernandez’s attack pattern template, as the attacker’s identity is essential in considering its intent, motivation and abilities</td>
</tr>
<tr>
<td>Motivation (why):</td>
<td>The purpose of the attack, which is different from the intent field that describes the desired effect</td>
</tr>
<tr>
<td>Methods (with what):</td>
<td>The methods, tools and resources used by the attacker. Also includes subjects when they are used as means to an end, as in social engineering attacks</td>
</tr>
<tr>
<td>Attacker skill (internal with what):</td>
<td>The skills needed to execute the attack, which are internal attributes using their own bodies (social engineering or physical attack) or enabling them to use external methods (hacking techniques)</td>
</tr>
<tr>
<td>Intent (what):</td>
<td>The intended results for the attacker. Our ontology separates out different types of result into ends and means by distinguishing between the ultimate incident effects and immediate stage effects. The attack pattern includes the logical effect (e.g. breach of confidentiality, integrity and availability) that could be utilized to achieve the social objective. The ultimate objective is not included within the intent field, but is considered within our wider incident model</td>
</tr>
<tr>
<td>Target (to what):</td>
<td>We include the type of target as a separate field. The desired result can only occur if the target is within the system and accessible to the attacker</td>
</tr>
<tr>
<td>Context or prerequisites (logical where and when):</td>
<td>Description of the conditions where the attack may occur. Includes the necessary functionality, characteristics or behavior that the system must or must not exhibit for the attack to succeed</td>
</tr>
<tr>
<td>Execution (how):</td>
<td>Known as solution in design patterns. Describes how the attack is performed to achieve its expected results. Our incident classification aids the determination of the logical progression and conceptual path followed. An attack tree with temporal sequencing may be included here to show incident progression</td>
</tr>
<tr>
<td>Process diagram:</td>
<td>One or more diagrams that visually show the attack execution and its participants. Class and sequence diagrams are common, and we also use our comprehensive incident diagram to show the entire hypothetical attack [4]</td>
</tr>
<tr>
<td>References:</td>
<td>Pointers to classification systems such as CAPEC, and to further sources of information about the attack</td>
</tr>
</tbody>
</table>

Figure 2 – Attack pattern template (abridged)

We indicate how our existing incident ontology helps to structure attack patterns making them more amenable to analysis and helping their formalization. We link the attack pattern template to the eight incident questions (shown in brackets) in figure 2, to illustrate how attack patterns help us to understand the surrounding incident context to gain a broader view of incidents.
The answers to the eight incident questions extend attack trees [14], which only model observable events (how) as links in the tree, to achieve the goal (what) at the root node.

The attack patterns contain generic fields or slots that are instantiated with particular entities when a matching attack is executed. Unlike typical design patterns, attack patterns generally give recommended methods of mitigating the attack, whereas we separate out the attack patterns from their corresponding defensive patterns. We omit several sections not related to our current discussion and omit the defense pattern template completely for brevity. Some of the new fields in the defense pattern are countermeasures and evidence, whereas many sections have analogs in the attack pattern.

3 MODELING ATTACK PATTERNS

3.1 Informal modeling advantages

Attack patterns attempt to provide some clarity, consistency and completeness in planning and implementing security solutions. They highlight the essential characteristics of the attack and therefore indicate possible defensive requirements. In our work, the attack pattern shows the undesirable activities, which aids the discovery of the corresponding defense patterns to mitigate the attack and its deleterious effects.

We have attempted to provide a greater degree of structure to attack patterns to aid their understanding and systematic analysis. Our security ontology provides a basis for an adequate semantics for attack patterns where the entities and relationships between them are suitably defined and understood. An ontology [17] represents knowledge as a set of concepts, their properties and the relationships between them within a domain of interest. The description and definition of the entities enables collective consistent understanding using the shared vocabulary, and may be used to reason about the domain.

We have seen that attack patterns suitably embellished with aspects of our security ontology, including especially the incident questions, can enable broader modeling of incidents. Conversely, attack trees [14] only indicate the incident actions and goals, whereas attack patterns also contain several other aspects such as motivation and surrounding context, enabling a wider analysis of defensive protection measures such as deterrence and changes to systems functionality.

3.2 Strategy for formalization

However, the current informality of these patterns means that their utility is limited to manual use, with little work in formalization to aid automation in creating effective defensive measures. We can use our security ontology together with Zhu and Bayley’s existing work on formalizing design patterns [18,19] to model attack patterns and their corresponding defense patterns in logic, which would aid better understanding and reasoning about the effectiveness of possible defensive measures.

The attack patterns form a specification for the attacker’s behavior and goals that must be recognized, understood and countered by the defense, using various defensive controls in network nodes and computers. A key issue is that the attacker’s activities cannot always be directly recognized, but must be inferred from indirect evidence, and so we separate attack patterns from their corresponding defenses patterns. The defense patterns form a logical specification for implemented defensive controls to counter attacks, and to collect adequate evidence for system recovery. We intersect the entire defensive pattern with the defensive network, computer and software capabilities at their particular vantage points within the system, to create an individual defensive pattern for each node that contains its view of the attack and response abilities from its specific location.

The attack patterns and corresponding defense patterns will be described in logic as specifications of possible states and behavior, aided by our linkage to the eight incident questions. A specification in logic that can be progressively refined into code is a common method of developing high integrity software. For example, a logical specification using the logic-based B method [20] can be progressively refined into executable code whilst maintaining correctness at each stage.

We intend to formalize the attack patterns from the Common Attack Pattern Enumeration and Classification (CAPEC) [13] overseen by MITRE. Other possibilities are Common Vulnerabilities and Exposures (CVE) [21] and Common Weakness Enumeration (CWE) [22], but CVE and CWE model defensive vulnerabilities and weaknesses respectively, and are therefore better at directly representing defensive patterns.

The reader may ask here why we do not simply create the required defensive patterns to model possible protection measures directly. This is certainly plausible and finds expression already in the idea of security patterns [23]. However, there is no direct linkage to possible attacks and so the defensive security patterns may be hard to validate. Conversely, attack patterns model real attacks and their formalization holds out the possibility of proving that some classes of attack must fail (assuming the defensive controls are operating correctly), or discover the conditions under which they can succeed to determine where additional defensive controls will be required.

3.3 Formalization

We describe the difference between logical proof and model-theoretic semantics [24 pp247-250], as it is relevant to proofs of security. Proofs refer to logical entailment between sentences in logic, whereas semantics deal with possible worlds of individuals and their relationships. The link between the two is that the arguments (constants like Alice and variables like x ∈ Employees) of logical clauses refer to individuals in these hypothetical worlds, and therefore logical proofs should make true statements in these domains. There should be a sound and complete correspondence between the two views, where a statement can be proved in logic if and only if it holds in the domain of individuals under discussion.

In mathematical logic, the identity or even existence of these individuals is irrelevant, and any model satisfying the soundness and completeness condition above will suffice. However, we wish these individuals and their characteristics and relationships to refer to real entities, which is why we use our security ontology. The bottom line, if the above paragraph is unclear, is that our ontology includes the relevant entities, properties and relationships, so that logical proofs about the patterns prove the required security properties hold for the particular entities involved in the defense of the system under discussion. This is why the fields in the attack pattern are represented using our security ontology, including the level, location, purpose, behavior and state of entities, rather than simply as a set of textual fields without adequate semantics.
We can view attack patterns as templates, with slots to instantiate the variables for particular attacks. One way to prove an attack pattern cannot succeed is to use the logic programming language Prolog to represent the pattern. The goal of the attack is then posed as a query to the program, which attempts to discover the circumstances where the goal will succeed. The query will fail if there is no possible attack with that pattern against the system under discussion, and will instantiate the variables if it succeeds to show all the circumstances where the attack is possible.

The formalization of the attack patterns thus aids translation into programs, but there are some differences from existing work on transforming logical system specifications into executable code. The defensive implementation must be divided into controls at different locations and levels with their particular recognition and response abilities, where we indicate in brackets how our existing ontology helps to overcome the issues including:

- Network devices, computers and applications do not have complete visibility of incidents or all the necessary powers to respond (our architectural model contains location) that helps to determine what attack actions can be observed by each node, and level helps to determine what events the node can understand
- Defensive controls may not see incident actions when required (our incident progression model divides incidents into stages that help to decide when the defense must be deployed)
- The protection measures may give indirect protection at a different level, location or time if the main attack vector cannot be detected or countered (aided by the translation of attack patterns to the corresponding defense patterns that contain the deployable defensive measures)

4 CONCLUSION AND FURTHER WORK

We outlined a roadmap for formulating attack patterns in our security ontology, which is useful for understanding and analyzing attack patterns. We then indicated our strategy for formalizing attack patterns in logic, which can possibly aid automated translation into code. We are modeling some of the CAPEC attack patterns. We plan to use the Web Ontology Language (OWL) [25] to give a well-defined semantics to these attack patterns, because we are using OWL to give adequate semantics to our security ontology. Then, we can use the description logic associated with OWL to model the patterns formally. We intend to report detailed findings in later work.

Existing ‘proofs of security’ often make a number of assumptions using unrealistic system models, where many systems that have been proved secure have subsequently been found to have serious flaw. We cannot prove any system is secure, without considering its surrounding context. We also need to consider the disposition of the system, its organization, critical assets, weaknesses, goals and adversarial threats, which are all included in our security ontology. Our formalization of attack patterns leads to the idea of secure-by-construction systems, where it may be possible to prove that certain attacks must fail, rather than give an unrealistic and all embracing proof of system security.

We focused on security in this paper, but similar ideas also apply to the creation of forensic patterns to determine the evidence that may be collected to hold the perpetrator responsible and repair damaged systems after a successful incident.

5 REFERENCES

A Heuristic Approach for Secure Service Composition Adaptation

Bo Zhou, David Llewellyn-Jones, David Lamb, Muhammad Asim, Qi Shi, Madjid Merabti
School of Computing and Mathematical Science
Liverpool John Moores University, UK,
Byrom Street, Liverpool L3 3AF, United Kingdom
{B.Zhou, D.Llewellyn-Jones, D.J.Lamb, M.Asim, Q.Shi, M.Merabti}@ljmu.ac.uk

Abstract—Secure adaptation of service composition is crucial for service-oriented applications. An effective adaptation method must improve a composition’s adherence to specified behaviour, performance and security guarantees at reasonable cost in terms of computing complexity and time consuming. This paper discusses current techniques that have been developed to help achieve secure service composition. Based on security verification results, which have been categorised into four patterns, a simple heuristics-based adaptation strategy is proposed. In order to make direct comparisons of different services, a novel quantification method is also introduced.

Keywords—composition adaptation; heuristic algorithm; security; service quantification

I. INTRODUCTION

A service-oriented architecture (SOA) provides the opportunity for services from different providers to work together. SOAs offer new applications via composition of services; facilitated by standardised interoperations among services. An important issue that shadows SOA platform and application development is security. Concerns around inconsistent security policies and configurations must be addressed and continually monitored with adaptation.

As part of the work undertaken for the Aniketos project we carried out a study on existing techniques relevant to secure service composition. The result reveals that while many efforts have been made in different areas to support SOAs, a practical solution is still missing for composition adaptation, which plays a crucial part in the secure service composition process.

In this paper we consider this issue in more detail and present an initial approach for composition adaptation. Instead of random substitution of services, we try to make the adaptation process more accurate and fast. It targets component service in a composition based on verification result. In order to compare services directly, a quantification method is also proposed to support the adaptation strategy. The rest of the paper is organised as follows. The next section presents an analysis of existing techniques in the area of service composition and verification adaptation. A heuristics based process is proposed in Section 3 towards secure composition adaptation. In order to support the heuristic method, Section 4 discusses a novel algorithm that quantifies services in terms of security. The paper concludes with a brief review in Section 5.

II. SERVICE COMPOSITION AND VERIFICATION

A. Service Composition

SOA platforms provide a foundation for modelling, planning, searching for and composing services. They specify the architectures required, as well as providing tools and support for service composition standards. Enterprise servers such as Glassfish or Microsoft IIS do offer security parameterisations, yet these are typically domain or platform-specific [1].

Therefore, in order to facilitate service composition across platforms; service modelling languages are used to describe a) the business requirements of a system, and b) system resources. By expressing behaviour processes and system organisation in agreed formats, compositions can be validated against desired criteria and modified to suit required changes in operation.

A number of languages relevant to service composition already exist. BPMN can be used to model relationships of composed services within a business model or process workflow [2]. WSDL is widely used for describing a Web service interface [3]; subsequent standards augment the basic description to add semantic, behavioural, and to a limited extent, authentication and security data [4]. Other such property-based extensions, including USDL [5], constitute standards that target trust and security, to bridge the previously-identified vendor divide. Some work tries to look at the issue from the point of view of QoS, i.e. achieve high productivity and low cost while producing new application [12]. Security-domain-specific modelling languages are also used to specify security requirements and policies. For example, S×C [6] is able to support programmable negotiation and monitoring of services with regard to agreed contracts.

However, selecting the most suitable service from a service pool can be problematic, especially when services are previously unknown. In such cases, trust plays an important role in service selection. S×C×T extends S×C, supporting maintenance of trust based on recommendations and penalties [7]. Trust calculation relies heavily on consideration of which parameters affect the trust value. The chosen parameters must be monitored closely to provide an accurate indication. In
paper [11], the authors proposed a formal specification-based testing approach towards accurate service selection.

B. Composition Verification Techniques

Once services are selected for composition, verification can occur to ensure selected services comply with security policy at both design-time and run-time. At design-time, formal analysis and proof-based techniques can establish that stated service behaviour is compliant. AVANTSSAR [8] is a tool for reasoning on service compositions. The Deploy tool supports formal analysis of security properties [9].

The Mobile Agent Topology Test System (MATTS) is a platform that has been designed primarily to allow modelling of system-of-systems scenarios and testing of secure service composition analysis techniques at run-time [13]. It supports analysis and verification of various security-associated risks for the composite services.

In MATTS there is a graphical user interface that allows creating different scenarios. A user can construct a scenario by specifying the used services and their relationships. The term ‘relationship’ here in can be considered in general terms to be abstract, with different relationships annotated using different properties. However, in this particular case the relationship represents the information flow between services. For example, if a navigation service A requires the input from a location service B, their relationship is simply represented as shown in Figure 1. The arrow represents the flow direction of the location information. In addition, the user can also specify the security properties of an individual service through the interface. These properties will be used to determine the overall security features of the composition.

As scenarios in MATTS allow evolution and change at run-time, the properties and connections between services are dynamically analysed to determine the security properties of the overall service composition.

Fig.1. User interface indication an information flow relationship between services.

Behind the user interface, there is a backend engine that analyses the scenario and its properties at run-time. Figure 2 provides a very brief overview of the security property analysis process. This is based on a special scripting language that allows the incorporation of both service properties and the interactions between services to be considered. For example, the script file can specify what kind of encryption algorithm a service should adopt, or which particular service is allowed to make connections. Each script file specifies a set of rules that determines whether the composition satisfies a particular security property. Once a script is loaded, the engine will measure the scenario against its rules. If the measurement result indicates a violation of a rule, the engine will notify the user. This process repeats every time when the scenario evolves.

Fig.2. The property analysis process.

As scenarios in MATTS allow evolution and change at run-time, the properties and connections between services are dynamically analysed to determine the security properties of the overall service composition.

III. HEURISTICS-BASED COMPOSITION ADAPTATION

Composition Adaptation aims to mitigate issues identified during verification and monitoring. Available actions normally include the following:

- Rearranging components.
- Adding or removing components.
- Enclosing components in a wrapper.
- Reconfiguring components (e.g. changing on parameters).
• Direct code adaptation (e.g. using aspect weaving).

Although adaptation options seem adequate, deciding the best action is complex. For example, adaptation can often be presented as an optimisation problem with constraints; however in certain security cases this has been demonstrated to be a NP-complete [10]. Consequently heuristic methods are likely to offer the only practical way to address these cases.

Heuristic algorithms solve problems based on experience, where exhaustive search for a solution is impractical. Adaptation of a composition falls into this category. Even though the verification process can tell a problem has been identified, it can nonetheless be computationally complex to give immediate suggestions as to how to correct the issue. For example a brute force approach to composition adaptation might simply cycle through and verify each of the possible solutions until a composition with adequate security characteristics is found. However, with the number of services getting large, this process quickly becomes unmanageable. In contrast, heuristic methods select compositions ‘intelligently’, attempting to find increasingly successful adaptations until an adequate solution is found.

For our initial approach we present a pragmatic technique that improves based on the level of detail returned by the verification process. In future work we aim to use methods such as genetic algorithms, swarm intelligence or simulated annealing to improve the optimisation process.

As discussed in the last section, verification techniques are available to establish whether a composition satisfies the security policy. The verification result may just be a Boolean value which is either true or false. Nonetheless, it is also not unusual for the verification result to include two other elements or any one of them: the violating service \(S\) and the violated security property \(P\).

Based on this, we propose an initial process for composition adaption that replaces the violating service with a preferable alternative and verifies the adapted composition again. The strategy used depends on the pattern of the verification results:

- If the verification result indicates both the violating service \(S_1\) and its property \(P\), we simply replace \(S_1\) with another service \(S_2\) that has stronger property \(P\) with the same functionality. The strength of \(P\) depends on the nature of the property.

- If the verification result only identifies the violating service \(S_1\) without knowing the exact cause \(P\), service \(S_2\) will be selected to replace \(S_1\), where \(S_2\) is chosen based on an overall assessment value of the services.

- If the verification result only specifies the violated property \(P\) without identifying the violating service, the service that has the weakest security property \(P\) in the composition will be replaced.

- If the verification result specifies neither \(S\) nor \(P\), the service that has the lowest overall assessment score in the composition will be replaced. An assessment methodology will be introduced in the next section.

As an enhanced solution, if the new composition still cannot pass the verification after adaptation and the result remains the same (i.e. the verification result indicates the same \(S\) and \(P\), the adaptation process will be extended to a wider range that involves services either having the second weakest property \(P\) or one hop away from the previously replaced service \(S\).

IV. Service Quantification

Security is a broad concept that includes many aspects such as confidentiality and privacy. One service may be stronger than another in terms of confidentiality; while it is also possible that the very same service has weaker protection for privacy. To tell if a service is better than another, we have to find a way to determine the security value of services.

The key to a quick and accurate analysis of service is to formalise the security property descriptions. A service description can cover various features about the service. For example, one feature may state ‘Communication with others is secured’, while the other one may state ‘My estimated availability ratio is 90%’. Semantically the features can be described in very different ways. Although some policy languages have been proposed to formalise service descriptions [14], the content of the actual description is still very open. Due to this lack of confinement, it makes the comparison of different services and their compositions can be very difficult. It is necessary to have a dictionary that can define the description of services, both semantically and quantitatively. For example, in the above cases the text descriptions can be translated to statements like ‘secured communication = true; (optional: encryption algorithm = 3DES)’ and ‘availability = 0.9’. The subjects at the left hand of the equations, i.e. ‘secured communication’, ‘encryption algorithm’ and ‘availability’ in this case, should be items from the dictionary, and their format/data type should have been determined in advance. This will allow a platform to be built that can measure and compare different services and their compositions without human intervention.

To judge if a service composition is more secure than another, we propose the concept of security impact factors to standardise security properties such as those mentioned above. For example, ‘secured communication = true’ could contribute 0.5 to Confidentiality and 0.3 to Privacy; ‘encryption algorithm = 3DES’ could contribute another extra 0.1 to Confidentiality. A security property could have impact on more than one key security aspect such as Confidentiality and Privacy. The security impact factors are predetermined based on expertise. With the dictionary that defines the security properties to be checked, the process of calculating security impact factors can be fairly fast. Moreover, in situations where new threats are identified or there is a breakthrough in the security protection techniques, the dictionary can help allow the security impact factors to be updated and maintained easily. At the end of the evaluation process the services will be quantitatively estimated in four key areas, namely Confidentiality, Authenticity, Availability and Privacy. These are also the main properties that are likely to be specified in a user's security policy.
Once the four security impact factors have been calculated we can give users the flexibility to adjust the weight of these four properties. This is important since in different scenarios the user's priorities may change. The weight could be any number between 0 and 1. Now assume a user sets the weights to 0.4, 0.2, 0.1 and 0.3 respectively for the aforementioned four properties, the overall security value $I$ for the service will be:

$$I = 0.4 \times C + 0.2 \times U + 0.1 \times V + 0.3 \times P$$  \hspace{1cm} (1)$$

where $C$ represents the value of Confidentiality, $U$ represents Authenticity, $V$ represents Availability and $P$ represents Privacy.

V. FUTURE WORK AND CONCLUSIONS

Based on our study, secure adaptation of composition is an area that demands significant effort for a feasible and practical solution. In this paper, we proposed a rather intuitive method that making adaptation of service compositions based on verification result. A supporting algorithm is used to quantify services in respect to their security properties.

There are many other factors that could affect the adaptation strategy. Nonetheless, we aim for the proposed method to serve as a starting point for achieving more effective secure adaptation of service compositions in the future.

The proposed adaptation strategy will be assessed in the Aniketos project, together with other strategies such as simulated annealing based approach. Their effectiveness will be tested based on adaptation speed and accuracy.

REFERENCES

An Overview of Artificial Intelligence based Pattern Matching in a Security and Digital Forensic context

Dr Faye Mitchell,
Department of Computing and Communication Technologies, Oxford Brookes University,
Wheatley Campus, Wheatley, Oxford OX33 1HX
frmitchell@brookes.ac.uk

Abstract – Many real world security and digital forensics tasks involve the analysis of large amounts of data and the need to be able to classify parts of that data into sets which are not well or even easily defined. Rule based systems can work well and efficiently for simple scenarios where the security or forensic incident can be well specified. However such systems do not cope as well where there is uncertainty, where the IT system under consideration is complex or where there is significant and rapid change in the methods of attack or compromise.

Artificial Intelligence (AI) is an area of computer science that has concentrated on pattern recognition and in this extended abstract we highlight some of the main themes in AI and their appropriateness for use in a security and digital forensics context. These themes are explored further in the full paper.

1. Introduction
In [1] we outline, from an academic and practitioner perspective, three of the main challenges in digital forensics:

1. the exponential growth in storage capacity, in single drives (hard drives, USB sticks, optical media, …)
2. the growth in distributed systems and the sophisticated forms of attack that can now be launched
3. the degree of technical sophistication employed by opponents and the apparent inability of existing tools and methodologies to keep pace [2]

In [3] we add a fourth challenge

4. the ubiquity of electronic storage and the range and prevalence of disparate storage systems

Although these challenges were originally specified with respect to digital forensics, they are also important challenges to be dealt with in the cyber security arena.

From a practical point of view however, these challenges all involve the requirement to deal with large amounts of rapidly changing information in a timely fashion. For cyber security, timely will normally mean in close to real time response as is possible. In digital forensics, there isn't this requirement for real time (although the time taken must be bounded), but the amount of potential data to be investigated is significantly larger.

2. Artificial Intelligence
For the sake of this paper we will use a pragmatic definition of AI as “a computer process or set of processes that acts in a manner that an ordinary person would deem intelligent”. Furthermore we will limit our discussion to the areas of AI that have direct relevance to cyber security and digital forensics and in particular we will focus on techniques that perform some aspect of pattern matching. We will not consider techniques such as robotics (as they have limited application to cyber security and digital forensics) nor we will consider techniques such as expert systems (which although technically classifiers are limited in their pattern matching scope).

3.1. Knowledge Representation
Knowledge representation is a key part of any AI system and is the means by which we represent both the knowledge and the reasoning about that knowledge. It also forms a key part in attack patterns as there needs to be a representation language for the concepts used in the patterns. From a cyber security perspective an ideal knowledge representation language has the properties of being inferential (you can reason using it), flexible & comprehensive (sometimes known as a languages representational adequacy), extensible, usable (normally this relates to how easy it is for humans to interpret) and community support.

The inferential property is simply a matter of making sure that the language is consistent and supports some sort of inference engine, and as such isn't difficult to achieve. Similarly, the usability property is normally a function of how close the language is to a conventional spoken language (yet still avoiding ambiguity) and is also not unique to the cyber security domain and is achievable. Flexibility, comprehensiveness and extensibility are normally addressed by using a lower level knowledge representation language which has those properties (e.g. XML or Ontolingua[4]).

Perhaps the hardest property to address is the most pragmatic of all, a languages support by the community that it is designed for. It is a well-known catch 22 situation that until a representation language receives buy in from the community it is aimed at it will not have sufficient support in the associated community and the tools they use for it to achieve it’s desired functionality.

There does exist some specific knowledge representations for cyber security [5] and digital forensics [6] and there are standard knowledge representations for specific technologies (e.g. Cisco ACLs, IP chains rulesets). However, these do not at present satisfy the requirements for an ideal knowledge representation language. [5] perhaps comes the closest in that it satisfies the inferential, flexible, comprehensive and extensible properties however it is debatable whether or not it satisfies the usability concept because of it’s complexity.
knowledge between tools and resources. It would also greatly facilitate the sharing of widely supported by the community and the tools used in the meets our ideal language requirements. In particular, if it was be used to generate a knowledge representation language that common concepts in an extensible format which could then ontology.

A domain ontology would provide standard definitions of common concepts in an extensible format which could then be used to generate a knowledge representation language that meets our ideal language requirements. In particular, if it was widely supported by the community and the tools used in the community it would greatly facilitate the sharing of knowledge between tools and resources. It would also facilitate the creation of a well described collection of test cases with known behaviours which could be used to test cyber security tools and techniques in a similar manner to what the UCI ML Repository \[8\] does for Machine Learning.

In digital forensics there is a recognition that such work to provide a domain ontology is required, but little has been undertaken so far. In cyber security, this is not as yet seen by many as a problem. However, without addressing it, many of the modelling or reasoning techniques that cyber security would benefit from can only be at best partially successful.

3.2 Pattern recognition techniques

Pattern recognition techniques are classifiers that allow the efficient description and recognition of sets of data that are of interest to the user, based on a description or statement of a certain pattern. A key feature of most pattern recognition techniques are the notions of generalisation and specificity. Suitable patterns must be specific enough to match all known cases of the set, and nothing that isn't in the set, but also general enough to recognise previously unseen examples of the set.

3.2.1 Machine Learning

The problem with naive pattern matching techniques is in the creation of the description of the patterns – often we can say X is an example of Y, but not be able to produce a suitable description that meets our generalisation and specificity requirements. (A good example is can be found in whiskey distillation where a master distiller can blend the different casks together to produce a fine whiskey but not be able to tell you why a particular set of casks should be combined.) Machine Learning (ML) is one of the branches of AI which tries to learn the classifications of interesting sets. ML techniques are generally thought of with respect to 2 dimensions – their representation format (symbolic or

1 OWL is a web markup language for creating ontologies. The term ontology is used to mean a shared vocabulary and taxonomy that can be used to describe the concepts and relationships in a given domain. The main difference between an ontology and a knowledge representation is that an ontology is designed to be shared, whereas a knowledge representation language is not.

3.2.1.1 Supervised Learners

Supervised Learning is the name given to the ML techniques (such as decision tree generators or Artificial Neural Nets) where the information is pre-classified into the relevant sets before learning occurs. This is useful when we have examples of the sets, but are unsure of how to describe the sets. This is perhaps the most appropriate form of ML for cyber security and digital forensics as it can provide accurate descriptions of the attacks that we seen on our systems, and even with subsymbolic learners (see 3.1.2.4) can give reasonable explainability. However it is not as good at adapting to previously unknown situations.

3.2.1.2 Unsupervised Learners

Unsupervised learners (such as conceptual clusters) are used when we do not know, or are unwilling to classify the available data into sets. This is particularly relevant when we are dealing with the large amount of data that is available when we are monitoring or investigating large systems. In general, the results given by unsupervised learners are not as accurate as those given by supervised learners for known issues but they can perform much better on previously unseen issues.

3.2.1.3 Symbolic Learner

Symbolic learners (such as Learning Apprentice Systems or decision tree generators) are ML techniques where the classifications gained by the learner are in a form that is relatively understandable by a human being and where distinct classification attributions can be critiqued and modified easily. Symbolic learners are therefore most useful when the results of a classification and/or action have to be explained to a human being. This is particularly of relevance in digital forensic investigations.

Symbolic learners are also much easier to tie into existing systems, both in terms of the knowledge based systems but also more conventional security systems such as firewalls. For this reason, symbolic learners, particularly supervised symbolic learners can provide a useful first step in adding intelligence into cyber security systems and could provide a mechanism for automatically generating cyber attack/defence patterns.

3.2.1.4 Subsymbolic Learner

Subsymbolic learners (such as Artificial Neural Networks or Hopfield Nets) are ML techniques where the classifications gained are in form of a weighted graph or other such notation that isn't easily comprehended by a human. These learners lack the explainability of their symbolic counterparts, but are often faster in use (although not to learn) and provide better generalisation characteristics.

Subsymbolic learners are also very good at learning in unsupervised situations where the initial classification of the training examples is unknown, approximate or difficult to produce. However, this coupled with the lack of explainability can lead to the subsymbolic learner learning a
used to play a significant part of a systems defence simply defending against unknown attacks they are less likely to be used as an autoassociative and a bidirectional associative of interest in a cyber security context is that they can often be learner which can act on the information.

3  An emphasis is on techniques which are not knowledge intensive ML techniques discussed in 3.2.1 form part of KD but the and relationships in extremely large (normally sparse) data useful in cyber defence.

These approaches suffer from the fact that they have no ability to detect a wide range of attacks however, both of completely different context. Together these can provide the subsymbolic learners ability to detect the same attack in a component. A bidirectional associative memory enhances a recognising the nature of the attack from just a small in detecting distributed attacks as it allows the possibility of using in cyber security situations, however they are good at examining historical data in detail and using that to find rules, relationships and classifications that can be used in a real time cyber security system. They can also be used to find attacks that might otherwise be below the noise threshold for more conventional ML techniques, in one instance detecting patterns that had previously been considered too small to be worth detecting [10].

3.3 Exploratory Data Analysis
Exploratory Data Analysis (EDA) is the combination of a suitable knowledge representation or ontology, a pattern recognition system and Data Visualisation (DV) so that a human can explore and understand the data under investigation. EDA is often seen as the poor man's alternative to ML or KD as it requires human input. However in cyber security, it is often perceived to be desirable to leave the final decision on the action taken to the human and to present the relevant information in such a way as facilitates that decision. This can also be thought of as a relatively easy way to add AI into a system as the computer can be given more mathematical and logical tasks and the bulk of the more complex reasoning can be offloaded to the human. (Which also has the advantage that the human is more likely to accept the final results as they trust their own judgement.)

A better use of EDA is to use the EDA to guide the discovery or learning process in KD and ML. This requires the ability to display high dimensionality data in a form the user can understand and process. Fortunately as [11] illustrates, the human perceptual system is able to process data with high dimensionality, in some case, if the data is presented properly, up to 25 simultaneous dimensions. When coupled by the humans pre-attentive focusing ability 3, this allows EDA systems to rapidly an accurately process large amounts of data.

3.2.2 Knowledge Discovery
Knowledge Discovery (KD) is the use of ML, probabilistic and statistical analysis techniques to discover classifications and relationships in extremely large (normally sparse) data sets. It can be viewed as a form of pattern recognition and the ML techniques discussed in 3.2.1 form part of KD but the emphasis is on techniques which are not knowledge intensive and are efficient enough to perform as these are tractable on large data sets (normally GB to PB in size).

Unfortunately, in any large collection of data there are going to be an extremely large collection of patterns and relationships, the vast majority of which are going to be of no use to the user. Therefore a feature of KD is the use of an interestingness metric, normally referred to as a J measure. This is a mechanism by which we can control the amount and type of classifications that we generate and limit them to ones that are potentially useful to us. The advantage of this is that we can greatly speed up both the learning of classifications and the detection by limiting ourselves to only that which is relevant. The problem is one of specifying a correct J measure and this is normally done on a per domain instance although there are standard methodologies to help generate it.

KD systems are normally highly interactive and time consuming to run. This obviously rules out their use in many cyber security situations, however they are good at examining historical data in detail and using that to find rules, relationships and classifications that can be used in a real time cyber security system. They can also be used to find attacks that might otherwise be below the noise threshold for more conventional ML techniques, in one instance detecting patterns that had previously been considered too small to be worth detecting [10].

2 This is best illustrated in a (possibly apocryphal) story about the US military who tried to train an ANN to recognise tanks hiding in trees. To this end they took pictures of forests with no tanks, pictures of forests with tanks and showed them to the ANN. Unfortunately the pictures without tanks were taken on a cloudy day and the pictures with tanks were taken on a sunny day so the ANN learnt how to tell if it was sunny or not. Because an ANN has no explainability power this fact was not found out until much later in the testing process.

3 An autoassociative memory is a memory system that can retrieve an entire data set based on just a small part of that data. A bidirectional associative memory is a memory system that can retrieve a related but different dataset.

4 Pre-attentive focusing is the name given to a humans ability to see patterns in apparently random data. The disadvantage of this is that humans can spot patterns when no pattern really exists.
3.4 Knowledge Refinement

Unfortunately, the patterns in any pattern recognition system can and will become outdated. In particular changes in the behaviour of a systems users can quickly result in pattern recognition systems generating lots of false negatives. One approach is to simply do the learning again and replace the existing patterns with the new learnt patterns. While this may be feasible for small cyber security scenarios, the effort involved in most cases renders it infeasible.

A better approach is to “patch” our knowledgebase of patterns. Unfortunately, manually patching such a complex knowledgebase is very error prone and automatically patching can be unpredictable and result in refining a knowledgebase to allow through previously blocked attacks.

One solution to this is to use a technique developed in the early days of Knowledge Refinement, that of a chestnut case [12], which is essential a test case which must succeed after the refinements have been made. This allows us to ensure that even after refining the knowledgebase, the main cyber security issues will still be detected.

4. Conclusion

AI techniques are very good at helping us find patterns in our data and as such has great potential for helping to solve cyber security and digital forensics issues. The difficulty is in selecting the appropriate tool for this domain and in this paper we have highlighted the main relevant approaches and commented on their appropriateness for the different aspects of cyber security and digital forensics.

This is by no means an exhaustive summary and there are many other possible AI techniques that might be applied to the domains of cyber security and digital forensics that we have insufficient space to discuss here (for instance, Support Vector Machines might be used to reduce the dimensionality of the problem, agent based systems could be used to better utilize distributed resources and to help defend against distributed attacks in large complex systems, conceptual clustering could be used to help identify important areas in the evidence under investigation and so on). Instead we have tried to concentrate on the more basic and common AI techniques could possibly, in one form or another, be used in the near future in the cyber security and digital forensics arena and to give a brief insight into the strengths and weaknesses of such techniques and to indicate where they might be used.

References

On Partitional Clustering of Malware

Renato Cordeiro de Amorim
Department of Computer Science and Information Systems
Birkbeck University of London
Malet Street, London WC1E 7HX
Email: renato@dcs.bbk.ac.uk

Peter Komisarczuk
School of Computing & Technology
University of West London
St Mary’s Road, London W5 5RF
Email: peter.komisarczuk@uwl.ac.uk

Abstract—In this paper we fully describe a novel clustering method for malware, from the transformation of data into a manipulable standardised data matrix, finding the number of clusters until the clustering itself including visualisation of the high-dimensional data. Our clustering method deals well with categorical data and clusters the behaviour data of 17,000 websites, acquired with Capture-HPC, in less than 2 minutes.

Keywords: Malware; Intelligent K-Means; Clustering.

I. INTRODUCTION

Malware is a popular term used to describe software designed to perform undesirable actions, such as gain unauthorised access to computers, exfiltrate sensitive data or simply disrupt the normal operation of computers.

Large amounts of malware are discovered daily, already approaching 10,000 in 2010 [18]. Such quantity of malware coupled with the possibility of financial gain have created a rather wide variety of attack/deployment strategies, giving birth to names such as viruses, worms, spyware, adware, etc.

Malware has created a whole new industry focused in its development or software tools for detection. As an example of the latter we have Capture-HPC [9], a high-interaction honeypot client. Honeypot clients work by monitoring the state of an unprotected network client normally using kernel callback mechanisms. They look for changes in the file system, registry, network activity and so on, filtering out common state changes through an exclusion list. When a state change is detected, suggesting a possible malware activity, Capture-HPC updates a log file containing comprehensive information regarding the state change, including a list of files and registry keys changed as well as launched processes and attempted tcp connections that may have happened.

The nature of Capture-HPC makes sure there are no false negatives except when Capture-HPC itself is corrupted. In order to increase the detection rate of Capture-HPC for our experiments, we have given it the additional capability of emulating the presence of ActiveX components. This way a malware will always detect as installed any ActiveX component it is trying to exploit.

We believe that organising malware into homogeneous clusters may be helpful to generate a faster response to new threats, and better understanding of malware activities. Clearly one can generate different clusters by taking different points of view, such as the malware binaries or its behaviour. We have opted for the latter and Capture-HPC has proved to be an excellent tool to gather the behaviour of malware [11, 24, 23], being able to detect a high amount of different malware related activities.

The idea of homogeneity is directly linked to similarity and by consequence to a distance measure. In clustering squared Euclidean metric is the most popular distance measure, but it is not the most appropriate when an entity, in our case websites, can have a high amount of features, the activities. Such scenarios define a high-dimensional space and these can be difficult to organise due to the so called curse of dimensionality [7].

We present here a fast method for malware clustering. Our method generates a data matrix from the behavioural data of malware, standardise this data matrix and find the number of clusters in a dataset by using intelligent K-Means [20]. We show that the number of clusters obtained by our method is correct by ratify this number by analysing the Hartigan index [14] and perform visual analysis of the clusters by using the 2 first principal components of the data matrix.

Methods based on the presence or absence of a particular malware activity can be quite difficult to cluster with a fast partitional algorithm such as K-Means. The reason for this is that the data is most, if not all, categorical. Nevertheless, our method clusters 17,000 websites in less than 2 minutes.

II. BACKGROUND AND RELATED WORK

Cova, Kruegel, and Vigna [11] empirically showed that most cases Capture-HPC could not detect a malware, it was because the malware was targeting a plug-in not present in the system. Taking this into account we have updated Capture-HPC to emulate the presence of any requested ActiveX component by using AxCmock 1.

The organisation of malware could follow a supervised, semisupervised or unsupervised learning approach. While supervised and semisupervised require labelled data stating that a particular behaviour should belong to a particular group, unsupervised which is also known as clustering, is a data-driven approach that does not require labelled data at all. Clustering attempts to learn the patterns in a dataset using solely the data itself and a distance measure.

1Our new version of Capture-HPC will be available at https://projects.honeynet.org/capture-hpc by summer 2012. AxCMock can be downloaded at http://code.google.com/p/axmock/.
Although supervised and arguably semisupervised algorithms tend to have better accuracy than unsupervised, their requirement of labelled data can be difficult to meet in certain scenarios. Labelling a statistically significant amount of data could require an impractical effort because of the high quantity of malware released everyday. Another issue is that this labelled data would have to be highly accurate, otherwise the algorithm could learn incorrect patterns and classify malware under the wrong groups. These facts made us opt for unsupervised learning.

In general, clustering algorithms can be divided into hierarchical and partitional. The former, in its more popular agglomerative form, generates clusters by merging entities or clusters and can be visualised through a dendrogram. Partitional algorithms generate a single set of labels for the entities and can be considerably faster than its counterpart. In both cases the granularity of the clustering is defined by the number of clusters in the dataset.

The use of clustering algorithms in datasets related to malware was introduced, to the authors knowledge, by Bailey et al. [4] using, as most of the literature, hierarchical clustering. Because of the large amount of data and the so called zero-day attacks, we consider that the speed of clustering is crucial and have chosen to use partitional clustering in this research.

K-Means [5, 19] is arguably the most popular clustering algorithm there is. K-Means partitions each entity \( y \in Y \), in our case websites, into \( K \) clusters around centroids \( C = \{c_1, c_2, ..., c_K\} \):

1) Assign values to \( K \) centroids \( c_1, c_2, ..., c_K \), normally \( K \) random entities; \( S \leftarrow \{\} \)
2) Assign each entity \( y_i \) in the dataset to its closest centroid \( c_k \), generating the clustering \( S' = \{S'_1, S'_2, ..., S'_K\} \).
3) Update all centroids to the centre of their respective clusters.
4) If \( S \neq S' \) then \( S \leftarrow S' \) and go to step 2.
5) Output the clustering \( S = \{S_1, S_2, ..., S_K\} \) and centroids \( C = \{c_1, c_2, ..., c_K\} \).

The above algorithm iteratively minimises the sum of the squared error over \( K \) clusters, we show the K-Means criterion in Equation (1).

\[
W(S, C) = \sum_{k=1}^{K} \sum_{y_i \in S_k} d(y_i, c_k)
\]

where \( d(y_i, c_k) \) is a function calculating the distance between \( y_i \) and \( c_k \). K-Means is a rather successful algorithm, its popularity is mainly due to its easy implementation, simplicity, efficiency, and empirical success [16]. One can easily find implementation of K-Means in popular data analysis software packages such as R, MATLAB and SPSS.

Due to its constant use, K-Means weaknesses are well known, among them: (i) it is a greedy algorithm. There is no guarantee its criterion will reach a global minimum, meaning that the final clustering may not be optimal. Although there have been attempts to deal with this issue, most notably the classical solution of swapping entities between clusters given by Hartigan and Wong [15], this is a very difficult problem as the minimisation of Equation (1) is a NP-Hard problem, we will leave this for future research; (ii) it requires the number of clusters to be known beforehand; (iii) the final clustering depends highly on the initial centroids given to the algorithm, these are normally found at random.

In a number of scenarios, including ours, the exact number of clusters \( K \) may not be known. The literature of clustering malware tends to use hierarchical clustering algorithms [4, 6, 26] seemingly because it is possible to run such an algorithm without knowing \( K \). However it can be difficult to interpret results when no granularity is set via \( K \), possibly generating clusters with no significance. Another issue is that hierarchical algorithms are known not to scale well. For instance, it may take 3 hours to cluster 75,000 [6] while our method clusters 17,000 in less than 2 minutes (see Section IV ). We find that there is a considerable amount of research effort in attempting to find \( K \) that could be used in malware datasets [10, 14, 17, 20, 21].

Regarding the weakness (iii), K-Means is a non-deterministic algorithm. It may provide different clusterings if run more than once, this characteristic raises the question of which clustering to use. A common solution is to run K-Means a number of times, generating a number of clusterings, and pick the clustering \( S^* \) which is the closest to the K-Means data model. \( S^* \) will be the clustering with the smallest \( W(S, C) \) given by the K-Means criterion, Equation (1). This approach does seem to work in a number of scenarios, but it can be very lengthy when dealing with high amounts of data.

In order to deal with weaknesses (ii) and (iii) at once, we have decided to use Intelligent K-Means (iK-Means) [20] due to its considerable success in different scenarios [2, 3, 10]. The iK-Means algorithm provides an heuristic initialization for K-Means based on the concept of anomalous clusters. An anomalous cluster is a cluster of entities that are far from the centre of gravity of the dataset. iK-Means iteratively finds each of these clusters and uses their centroids and number of clusters as the parameters for K-Means. The algorithm is formalised below.

1) Assign a value to \( \theta \); set \( c_c \) as the centre of gravity of the dataset; \( C_t \leftarrow \{\} \)
2) Set a tentative centroid \( c_t \) as the entity farthest away from \( c_c \).
3) Apply K-Means using two centroids, \( c_t \) and \( c_c \) generating the clustering \( S = \{S_t, S_c\} \).
4) If the cardinality of \( S_t \geq \theta \) then \( C_t \leftarrow c_t \), otherwise discard \( c_t \). In any case, remove \( S_t \) from the dataset.
5) If there are still entities to be clustered go to step 2.
6) Run K-Means with the centroids in \( C_t \).

To demonstrate the method works with malware data we have chosen to ratify the number of clusters it finds with visual inspection and the Hartigan index [14] mainly because of its ease of use and popularity. This index is based on the error
 experiments [12, 25] show that distances such as the cosine ing an appropriate distance measure for K-Means. Although high-dimensional spaces. This can be accomplished by select- the dimensionality of the data increases [13, 1, 8].

Unfortunately the malware datasets are very likely to be large in terms of websites and measurements, visibly high- dimensional datasets. The curse of dimensionality, a term coined by Bellman [7] states that as the number of dimensions increases so does the sparseness of data making entities to appear dissimilar, a very problematic fact to distance-based algorithms such as K-Means. This is further supported by research suggesting that the concept of nearest neighbours algorithms such as K-Means. This is further supported by research suggesting that the concept of nearest neighbours calculated using Euclidean distance becomes meaningless as the dimensionality of the data increases [13, 1, 8].

In order to cluster malware we need a method that supports high-dimensional spaces. This can be accomplished by select- ing an appropriate distance measure for K-Means. Although the literature tends to use the Euclidean distance this is not the most appropriate in high-dimensional spaces. Empirical experiments [12, 25] show that distances such as the cosine are more appropriate than the Euclidean distance. The cosine distance for the N-dimensional x and y is defined as:

$$d(x, y) = 1 - \frac{\sum_{n=1}^{N} x_n \cdot y_n}{\sqrt{\sum_{n=1}^{N} x_n^2 \cdot \sum_{n=1}^{N} y_n^2}}$$

A somewhat easier way to apply the cosine distance, is to perform an extra step in the pre-processing of data by dividing each row vector \( y_i \) representing a website by the vector’s norm \( \sqrt{\sum_{n=1}^{N} y_{in}^2} \) [13, 22, 27]. We find this particularly helpful to calculate centroids of each cluster.

As final consideration for this section, a clustering algorithm regardless of being partitional or hierarchical, will not yield that a given cluster is composed of malware. Clustering algorithms simply find that 2 or more clusters are dissimilar according to a given distance measure and cannot state what they are actually composed of. In order to define what malware family a cluster contains one would need the analysis of a field expert. Clearly this expert would not need to analyse the whole cluster but solely the malware that is the closest to the cluster centroid.

III. METHOD

In order to apply any clustering method we need to create and standardise a data matrix representing the whole dataset \( Y \). In this data matrix each instance of \( Y = \{ y_1, y_2, ..., y_N \} \) represents a website and each column \( v = \{ 1, 2, ..., M \} \) a feature. In our method \( y_{iv} \) may be assigned 0 or 1 representing the absence or presence of a particular malware behaviour, a feature, in the website \( y_i \).

The first step of our method is to apply Capture-HPC on each website, recording its activities in a log file. The amount of time used for recording stays as a parameter, but we suggest it should not be less than a minute as some websites may take time to load. The full list of activities these websites performed on an unsecured network client becomes our initial list of features.

The second step is to filter the list of features. Although Capture-HPC uses an exclusion list to filter out expected state changes, it records data such as time and process ID generating groups of features that represent in fact the same malware activity. We disregard any part of the log file that is not directly linked to the activities, but to management, such as timestamps, process IDs, paths to files and IP numbers in tcp-connections. This filtering ensures each feature is unique in terms of what states the malware is changing, effectively reducing the number of features.

The third step is to create the data matrix, by assigning a value to each \( y_{iv} \), and standardise it. When creating the entry \( y_{iv} \) in the data matrix we read the log file for this particular website and search for of the \( M \) features we have listed. If a feature \( v \) is found to be in the log file then \( y_{iv} \) is set to 1, otherwise 0. Each value in our data matrix is categorical, making its standardization less obvious. We have opted to use a method presented by Mirkin [20]. In this each feature is transformed into 2 new features (since we have only 2 possible categories). Only one of the new features is assigned 1, the new feature corresponding to the category in the original feature, the other is assigned 0. We then standardise the data numerically by subtracting each of the values \( y_{iv} \) by the new feature average \( y_{iv} \), linking the final value of \( y_{iv} \) to the frequency of \( v \) over \( Y \).

In the forth step we apply the Intelligent K-Means algorithm with \( \theta = 1 \). We then sort the clusters in descending order by the number of websites found in each of them, in the anomalous cluster part of iK-Means. We choose the number of clusters by analysing when the cardinality of clusters stabilizes in a relatively small value.

Optionally, one can also ratify the number of clusters by using for instance the Hartigan index and the less reliable visual inspection. The former analyses the differences of the K-Means criterion shown in Equation (1) under different values for \( K \). Regarding visual inspection, we have chosen to plot the whole dataset over its 2 first principal components. Because the data is categorical in nature the clusters structure can be rather difficult to see. One may wish to add a small amount of uniformly random noise to help the visualization of cluster cardinalities.

IV. EXPERIMENTS

We acquired a list with 17,000 possibly infected IP ad- dresses\(^2\). In the first step of our method we applied Capture- HPC for 2.5 minutes to each of these websites, generating a

total of 133,327 features. By using our second step we were able to reduce the number of features to 231, making each feature truly representative.

We then created the data matrix of initially 17,000 websites over 231 by checking the logs of each website against the features we found. We finalised the data matrix by standardising it using Mirkin’s method, effectively doubling the number of features.

In the next step we applied intelligent K-Means with \( \theta = 1 \), finding a total of 15 clusters with more than 1 feature. We then sorted the clusters by the number of entities found in the anomalous part of the algorithm and found the cardinalities 8439, 7851, 381, 129, 121 and very small numbers after these, suggesting the dataset had 5 clusters.

The popular Hartigan index helped us to ratify 5 as the number of clusters. After the 5th cluster the error given by Equation (1) ceased to be significant. Although less reliable, we also found 5 clusters in this dataset by analysing visually the 2 first components of the dataset, extracted by using principal component analysis (PCA). The image can be seen in figure (1), to which we had to add random noise between 0 and 0.25 to increase clarity. It seems to us that figure (1) presents 2 clusters at the top, the one at the right being a bit more elongated, and 3 at the bottom, totally 5 clusters.

![Fig. 1. A Dataset containing the behavioural data of 17,000 possibly infected websites over its 2 first principal components.](image)

V. CONCLUSION

By using our method we were able to cluster a dataset of 17,000 malwares in less than 2 minutes. This was a particularly interesting database to work for 3 reasons: (i) the features were solely categorical; (ii) the cardinality of the clusters was uneven and (iii) it was a high-dimensional data matrix. Because of these three reasons the ordinary K-Means algorithm would constantly fail to cluster the dataset, generating empty clusters.

We solved the above issues by standardizing the data by transforming each feature into 2 new features allowing us to standardise them numerically through their frequencies in the dataset. We have also used the intelligent K-Means method to find the number of clusters in the dataset as well as the initial centroids for each of them. Finally, the issue of high-dimensionality was dealt by using the cosine distance rather than the more popular squared Euclidean distance.

This research assumes that the features obtained with Capture-HPC are all relevant. This sounds realistic since Capture-HPC uses an exclusion list for expected state changes, not leaving any reason why a client should suffer a change in state while not being used. Of course relevant features may have different degrees of importance for clustering particularly at different clusters. We intend to further developed a previous method used to find features weights called intelligent Minkowski K-means [2] so we can apply it to the clustering of malware.

VI. ACKNOWLEDGEMENTS

The authors would like to thank Tiffany Youzhi Bao for her instrumental work developing AxMock ultimately allowing us to upgrade Capture-HPC to mock ActiveX components.

REFERENCES


14. JA Hartigan. “Clustering algorithms. 1975”. In: *John Willey & Sons* ()


Where has this hard disk been?: Extracting geospatial intelligence from digital storage systems

Harjinder Singh Lallie¹, Nathan Griffiths²

¹International Digital Laboratory (WMG), University of Warwick, Coventry, CV4 7AL
²Department of Computer Science, University of Warwick, Coventry, CV4 7AL
{h.s.lallie, Nathan.Griffiths}@warwick.ac.uk

Abstract - Digital storage systems (DSS) contain an abundance of geospatial data which can be extracted and analysed to provide useful and complex intelligence insights. This data takes a number of forms such as data within text files, configuration databases and in operating system generated files - each of which require particular forms of processing. This paper investigates the breadth of geospatial data available on digital storage systems, the issues and problems involved in extracting and analysing them and the intelligence insights that the visualisation of the data can provide. We describe a framework to extract a wide range of geospatial data from a DSS and resolve this data into geographic coordinates.

I. INTRODUCTION

Digital storage systems (DSS) contain an abundance of geospatial data (sometimes referred to as geolocation data). Geospatial data is any data that can identify a geographic location. This data is stored in a number of locations and formats in a digital storage system, such as EXIF-data (Exchangeable Image format) within images, the photographic images themselves if they can be resolved to particular locations and as zip-codes/postcodes within contact databases. Geospatial data can serve a number of benefits if processed and analysed correctly and can be useful to enterprises, marketing agencies, law enforcement agencies and the defence services.

Research has previously taken place into the value of geospatial data in other domains, for instance the value of geospatial digital intelligence in the business/marketing domain has been highlighted by Stanhope [2] who charts the progression of web analytics from simple web server log analytics, through more formal web analytics, digital analytics through to digital intelligence. In this context companies are able to ‘track’ traffic and in particular the source of that traffic to create intelligence which assists in influencing business/marketing decisions and aiding ‘know your customer’ (KYC) solutions.

Geospatial data has proven valuable in medical research to identify cancer care and support needs and the way it links with socioeconomic, demographic and environmental factors and in particular the support needs within particular geographic areas [3-5]. Geospatial data generated by GPS (global positioning satellite) systems is used in a variety of applications including movement monitoring in sports [6, 7], monitoring movement of children [8], parcel/courier tracking [9], clock synchronisation [10], tracking sex offenders and bailed/paroled prisoners [11], in differentiated/mobility pricing to calculate insurance or road tax premiums [12-14] and perhaps most popularly in Satellite Navigation (SatNav) based systems [15].

In the area of law enforcement, geospatial data serves a useful purpose for Government for the analysis and publication of crime statistics [16], it is used by regional police forces to highlight crime rates in given postcode areas [17] (see Figure 1) and is also used by American states for systems designed to publish addresses of sex offenders [18].

One area in which more research can be conducted is in analysing, extracting and processing a wider range of geospatial data from a DSS and in particular extracting knowledge of underlying social networks and digital intelligence of criminal homophily. This paper describes such a framework and explores the process involved in resolving this data into geographic coordinates which can be used in subsequent applications. We use the example of criminal homophily to demonstrate the value of such a system.
II. GEOCODING

Geospatial data exists in one of two places on a DSS – as document data or application/configuration data.

Document data is textual references to places within user documents created in applications such as Word, Excel, Adobe Acrobat; in contact databases such as Microsoft outlook or data entered into online forms (stored as file fragments and cache). Within this category we also include photographic images which can be resolved to identify locations. The format of such data can be inconsistent, ambiguous and in many cases incorrect.

Application/Configuration data is data that has been input as part of the configuration process for an operating system, application or hardware to work, or which may be generated automatically by the operating system, an application or hardware such as a GPS receiver. Examples are: IP address information in particular applications and hardware, DNS data in web servers, email addresses, mail server details, cookies, cache generated by internet browsers and geo-tag data automatically stored in EXIF data in photographic images. An example of very precise geospatial data in this context is that generated by GPS systems in satellite navigation applications wherein precise coordinates are generated to demonstrate the receiver position and calculate the route. Application/Configuration data is structured to meet the standards of the application and is more easily resolved than document data.

Approaches towards geospatial data extraction from digital storage systems vary according to the location of the data on the storage system. Geospatial data may be located in specific, formally and precisely defined areas of files, such as records in databases or entries in operating system specific files (such as the registry in Microsoft Windows). An example is EXIF data, which can include geographic coordinates stored as metadata in images. This data represents precise geographical coordinates of the location where a photograph is taken. An example of this is shown in Figure 2. Similarly, email client applications store mail server details, web browsing applications may store local DNS details and operating system specific configuration files and databases may store IP addresses and other geospatial data.

Before geospatial data can be used it has to be translated into coordinates, through a process known as geocoding that involves one or both of two stages: resolution and correlation.

A. Resolving geospatial data

There are at least three instances where geospatial data needs to be resolved before it can be correlated. This is where the geospatial data is circumstantial/textual, is ambiguous or is a photographic image. In certain cases it may be necessary to resolve the data because of a combination of factors, for instance in the case of a photographic image which looks very similar to multiple locations or a misspelt location name which first has to be ‘corrected’ before it can be resolved to one of a number of geographic existences of that place name.

- Textual data. Textual data has to be searched for the existence of geospatial data which then needs to be matched with a reference dataset of place names in order to establish that it is indeed a place name. This process of geo-parsing is well covered in the literature, and as far back as 1966 systems such as GeoRef were based on textual analysis and deployed to create geospatial systems for scientific use [19]. The growth of the
internet further popularised research in this area. In 1994, Woodruff and Plaunt proposed GIPSY, a system which took parsed textual input to extract geographic terms and applied spatial reasoning and a degree of statistical analysis to arrive at a geo-position [20].

- **Toponym Resolution.** Occasionally a place name may be ambiguous because, for instance, it refers to more than one location that has the same name. As an example, Sheffield points to 28 geographic locations with most of them being in the USA [21]. In this case we must perform toponym resolution wherein through some contextual analysis we are able to better determine which instance of Sheffield the geospatial data refers to.

- **Photographic images.** Photographs may contain iconic landmarks which are in their own right geospatial data which can also be resolved automatically. This can be done through a process known as scene matching which was described by Hays and Efros and implemented in a system referred to as IM2GPS [22].

### B. Correlation

The geocoding system receives input data such as an address, postcode or trilateration metrics (as in the case of GPS systems), applies a feature/pattern matching algorithm to this and correlates the input data with geographic coordinates present in a geospatial reference dataset (a gazetteer such as the US TIGER/Line Shapefiles [23]). The reference dataset contains data which allows for the coordinates to be plotted on a particular point of the map, i.e. it correlates the input with a point on the map.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Processing features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geospatial data in images</td>
<td>Directly accessed from the appropriate field, high level of accuracy and requires no further processing</td>
</tr>
<tr>
<td>IP Addresses</td>
<td>Must be interpolated back to the source in order to be used. May require further analysis before it can be used due to a potentially higher degree of ambiguity</td>
</tr>
<tr>
<td>Addresses in textual documents</td>
<td>Requires geo-parsing possibly followed by toponym resolution</td>
</tr>
<tr>
<td>Addresses in contacts databases</td>
<td>Directly accessible from appropriate fields in databases, may require geo-parsing possibly followed by toponym resolution</td>
</tr>
<tr>
<td>Telephone number prefixes in contact lists</td>
<td>Directly accessible from appropriate field. Needs to be interpolated to provide a geographic range. High level of geo-granularity</td>
</tr>
<tr>
<td>Routes calculated and/or followed by users of satellite navigation system</td>
<td>Often stored as coordinates so very precise, rarely requires further processing</td>
</tr>
</tbody>
</table>

Table 1. Range precision of Geospatial data

In particular circumstances, multiple reference datasets may be used to increase the accuracy of the correlation process. However this also has contains the risk of introducing further ambiguities as the correlation process may present conflicting coordinates, in such a case, a best match can be calculated to return the most ‘likely’ or accurate set of coordinates [24].

Some of the features described herein are automated in online mapping systems such as Batchgeo, Google Maps, MapPostcode and Freemaptools [25-28] and the Geocoding system tends to be part of a larger system known as a Geographic Information System (GIS)/Geographic Information Retrieval (GIR) System.

The result of this process is either:

- A pair of specific longitude/latitude coordinates presented (for instance) to 8 decimal places which is accurate to within 1.11mm of a physical geographic location, or
- Or a group of longitude/latitude coordinates which point to a range (such as a postcode) that may cover an area of a few hundred square meters.

We can see from this that the resulting coordinates may refer to a single point or a geographic range, and we refer to this as geo-granularity. The issue of geo-granularity is important and should not be confused with accuracy. A street name which produces a set of coordinates which point to the entire geographical range of that street, a business name which the application plots precisely to the business address or a set of longitude/latitude coordinates which are plotted precisely to that geographic coordinate are all accurate. However the plotted geographical range is different in each case. That said, there certainly are issues of accuracy and errors can be introduced during the process of geocoding. Methods of improving the accuracy of plotting have been proposed by Bakshi [29] and Goldberg and Cockburn [24].

Of the types of geospatial discussed herein, IP addresses
present the greatest difficulty. Whilst IP addresses can provide a good indication of geospatial locality, they are also prone to a number of problems, for instance, IP addresses can be spoofed, they may point to the ISP instead of a user, they may be dynamically allocated and have other features which make it difficult to map more precisely. Further research needs to focus on how IP addresses can be used to provide more accurate geospatial locality. One method may be to use other contextual data to add value to the IP address.

III. THE VALUE OF GEOSPATIAL ANALYSIS

The resulting data has applications in a number of fields, and we use the example of criminal homophily to demonstrate this. Geospatial data can be extracted from digital storage systems to create what Bell refers to as a *forensic information pool* [30], which becomes an important source of intelligence in criminal investigation casework. The benefits that such a forensic information pool might present have not hitherto been explored. Some of the benefits may include:

- An analysis of geospatial data in the context of image hash datasets such as Childbase [31] to highlight geographic criminal trends and provide more efficient victim identification.
- If we treat geospatial data on a DSS as nodes, the geospatial data can be analysed to highlight social structures which show the density and intensity of communication between nodes, thus highlighting communication behaviour. This can in turn help to identify criminal homophily and further insights into geographic patterns in offending behaviour.
- Such systems have benefits outside of law enforcement as well. More detailed analysis of geospatial data can give better indications of consumer behaviour and habits.

IV. CONCLUSIONS

Based on the discussions presented herein we can summarise the main elements of this paper in Table 1 which describes the range of geospatial data and the issues relating to processing the data therein. We can see that particular data types require a greater degree of processing.

The resulting framework is illustrated in Figure 2. An input (address, telephone number, IP address etc.) is analysed and considered for resolution. It may need to be resolved if it is ambiguous. The input (or resolved version thereof) is correlated against a reference dataset using a *feature matching algorithm*, which results in a set of coordinates. As an example, if the input data is a UK postcode, this may not need to be resolved and can proceed to the correlation phase. The reference dataset in this case may be a UK postcode gazetteer which would return a set of coordinates. However, a textual reference to a place name would first need to be identified as a place name followed by further resolution to a set of coordinates.

The main contribution of this paper has been the synthesis of a number of geospatial data translation methodologies into a single framework and the consideration of the complexities and issues in correlating the various geospatial data into a meaningful set of coordinates.

The next step of this research is to implement this framework into a system which can ultimately be designed as an application.

REFERENCES


Attack Pattern Recognition through Correlating Cyber Situational Awareness in Computer Networks

Noor-ul-hassan Shirazi, Alberto Schaeffer-Filho and David Hutchison
School of Computing and Communications
InfoLab21, Lancaster University
Lancaster, UK
E-mail: n.shirazi@lancaster.ac.uk, asf@comp.lancs.ac.uk, dh@comp.lancs.ac.uk

Abstract—There is no denying that communication networks, in particular the Internet, have changed our lives in many ways. Many organizations and businesses in general benefit, but at the same time their communication networks face many challenges such as cyber-attacks, which can result in disruptions of services and huge financial losses. Therefore, resilience of these networks against cyber-attacks is a growing interest in the cyber security community. In this paper, we propose a framework for attack pattern recognition by collecting and correlating cyber situational information vertically across protocol-levels, and horizontally along the end-to-end network path. This will help to analyze cyber challenges from different viewpoints and to develop effective countermeasures.

Keywords—Multilevel Resilience; Attack Patterns; Pattern Recognition; Cyber Situational Awareness

I. INTRODUCTION

Communication networks are nowadays considered to be critical infrastructures [1]. Unfortunately, the risk of these networks failing as a result of cyber-attacks has also increased dramatically. Resilience, the subject of this paper, is the ability of the network to maintain normal levels of operation in the face of many types of challenges, including (D)DoS and other types of cyber-attacks, operational overloads, and mis-configurations [2]. Countries around the world are dedicating their resources to combat cyber challenges but they all agree that it is becoming increasingly difficult [3] as cyber criminals are becoming more organized and sophisticated. One example of such an attack is GhostNet – which has infected large numbers of computers in many countries, of which close to 30% can be considered as high-value diplomatic, political, economic, or military targets.

Moreover, network data has become more accessible to new tools and technologies that provide information about network monitoring and applications behavior. Several research efforts have focused on the development of resilience detection and classification techniques to build cyber situational awareness based on this information. The majority of tools to monitor networks and applications, however, perform analysis on data from single sources, typically from diverse and independent sensors that monitor network and application events. This makes the assumption (to be validated), that highly coordinated and sophisticated attacks are actually reflected in correlated patterns.

These datasets are typically available in the form of logs, which can be normalized and correlated to extract interesting evidence about attack patterns. Patterns provide a convenient way of encapsulating and reusing knowledge. Their purpose is to communicate proven solutions in a particular domain. The use of patterns has emerged from architecture work and has been applied to software engineering by Gamma et al. [4]. Our resilience requirements suggest that a focus should be placed on the process for layered defences by collecting and correlating cyber situational information vertically across protocol levels, and horizontally along the end-to-end network path. This has led us to the following initial questions:

- How should we define attack patterns?
- What classification schemes for attack patterns can we suggest?
- What success criteria can we propose for evaluating attack patterns?

In working towards answers to all the above questions we intend to extract meaningful information from multiple datasets using correlation of, for example, IP addresses, timing, and behavior (e.g., under (D)DoS attack behavior could be the ratio of numbers of packets of server ports to total number of all service ports, flows per second per interface etc.). This makes the assumption (to be validated), that highly coordinated and sophisticated attacks are actually reflected in correlated patterns.

Correlation is used to describe the process and results of establishing important connections between various data items from multiple sources, typically from diverse and independent sensors that monitor network and application events. This meaningful information, which we expect to see in an attack patterns, is intended to highlight serious threats in real time or at least in post-event assessment for adopting and designing overall resilience controls for networks.

This paper has the following structure: Section II presents related work. Sections III and IV explain the proposed model and our approach respectively. An example scenario is illustrated in Section V. Section VI discusses briefly the future work and finally concludes the paper.

II. BACKGROUND AND RELATED WORK

We aim to develop an effective and systematic method that can help to determine cyber-attacks consisting of a large number of correlated events. Attack vectors are usually largely distributed over the Internet and their lifetime varies, which can make them difficult to detect. Due to this nature, analyzing attacks is a challenging task. In our approach, we will use datasets for analysis with the assumption that high-level coordinated attacks will be reflected by the correlation of various attack events that originated from different parts of the network. Those correlations are normally not visible by using only individual datasets. We will use specific data mining and knowledge discovery algorithms, in order to assist us to find interesting attack patterns. Currently we are evaluating different clustering approaches such as Hierarchical, K-means and Graph based clustering [5, 6, 7] and comparing their performance and suitability in order to achieve objectives for our proposed model.

Traditionally, the process of attack detection and classification has been investigated by using the analysis of individual datasets, for example, NetFlow records, Server logs, and Web IDS logs etc. [8, 9, 10]. However, applying these classification techniques to a single dataset is not sufficient for identifying complete attack phenomena. Furthermore, since attacks evolve over time we believe that applying those techniques on a single dataset would not yield complete observation about specific attacks. The approach proposed in this paper is different in the sense that we aim to investigate how evidence from multiple datasets can be correlated, in order to achieve a more robust attack detection and classification.

An attack is an act of carrying out an exploit of vulnerability [11]. In [12], the author has developed a description and classification scheme for attack pattern. This scheme includes:

- Name and unique ID of the attack pattern
- Related weakness and vulnerabilities
- Method of attack and attack example
- Describing information
- Related attack pattern

The MITRE Corporation provides a publically available catalogue of attack patterns known as CAPEC – Common Attack Pattern Enumeration and Classification. CAPEC (which was established in the year 2000) is sponsored by the U.S Department of Homeland Security and technically managed by Cigital Inc. The attack pattern concept from CAPEC represents a description of common attack approaches abstracted from a set of known real world exploits. The catalogue provides classification for attack patterns along with comprehensive schema and classification taxonomy. As part of CAPEC, the CybOX – Cyber Observable Expression is provided which is a standardized schema and is written in XML – Extensible Markup Language. CybOX is a standardized language for encoding and communicating high-fidelity information about cyber observables, whether dynamic events, or stateful measures that are observable in the operational cyber domain. CybOX provides a common structure and content mechanism for addressing cyber observables across and among different use cases for overall cyber situational awareness.

In CAPEC, attack patterns can be seen as formalized textual descriptions of high-level attacks in terms of their characteristics, such as system vulnerabilities, weaknesses, and methods for exploiting software. To some extent, CAPEC seeks to define attack patterns using a top-down approach, i.e., cataloguing attacks and identifying their key elements from the attacker's perspective. Instead, we focus on the gathering and correlation of evidence from multiple datasets in order to identify patterns of correlated events that could explain an attack. Thus, we see the two approaches as complementing each other, in the sense that we will be able to identify and extract attack features, which could be subsequently referred to as observables in CAPEC's attack patterns.

Another work similar to ours is honey net traffic analysis, which aims to determine the occurrence of an attack by analyzing attack events using honey-pot traces. Our analysis will be different from any traditional analysis of traffic traces because we will be using spatial distribution and model the behavior of attack found in different correlated events, from multiple datasets.

Botnet tracking is another active research area relevant to our work. In [13], the author has developed a botnet detection framework based on clustering of C&C communication and activities flows to detect similarity patterns and combination of both types of patterns by cross-correlation. However, this work is different from ours as we aim to develop more general models that can be applied to the detection and classification of a range of cyber-attacks as opposed to a specialized technique that is targeted at a single type of attack.

Correlation has also become the most important technique for network management. Without applying this technique, the arrival rate of network events often becomes overwhelmingly high and limits the ability of the operator to investigate and respond to challenges [14]. However, currently, event correlation is mainly used for network management and we aim to extend this to other domains such as cyber situational awareness across multiple levels. In [15], the authors have implemented the Spice event correlation engine for detecting port scans, GrIDS [16] uses graph-based correlation for detecting intrusions, and Snort [17] also applies event correlation for detecting multiple intrusions. Nevertheless, all these correlation techniques are specific to single datasets and do not provide complete insight by incorporating heterogeneous correlation across various levels.

There are several other well-known projects that are used to monitor all unsolicited traffic directed to dark subnets. Darknets are unused IP subnets, and many projects operate darknets – such as the Team Cymru darknet [18] and the Internet Motion Sensor [19]. However, they are primarily used to analyze specific phenomena that are essentially related to worm propagation [20, 21]. Our approach will be different in terms of technique and objective, as we do not focus on the analysis of Internet worms alone.

Moreover, in [22], the authors have tried to characterize the nonproductive network traffic. They analyzed temporal patterns
and correlated activity within the unsolicited traffic. Recently, similar research has been performed by Arbor Networks [23]. Although, all this previous work provide meaningful results, at the same time they are not sufficient for complete cyber situational awareness. Therefore, we aim to apply more appropriate techniques in order to elevate the abstraction level, and to improve the insights into the behavior of global cyber situational awareness.

The approaches mentioned above are related to our work and they all provide building block techniques for analyzing threats. The model proposed in the next section intends to build on these techniques in order to support data analysis by combining collective intelligence, vertically across protocol levels and horizontally along the end-to-end network path, from different viewpoints. In summary, we note that most approaches for analyzing malicious activities are tailored to specific issues by means of particular datasets. However, these are usually only small pieces of the overall puzzle, which are only able to provide an incomplete picture of cyber activities.

III. MODEL

Traditional models describe how to cope with random challenges [9 10]. However, we aim to extend these models to deal with highly sophisticated attacks across the multiple levels in networks in order to provide deterrent controls against such attacks. In today’s communication network environments, events and alarms come from multiple independent sources and not all events are ordered and analyzed together, so information is lost and non-determinism is introduced [24]. This gives rise to the following further research questions:

- How can we identify a correlation of data that corresponds to an attack, based on the combination of all available evidence?
- How can we analyze the cyber-attack phenomenon from separate viewpoints, revealing different correlation patterns?
- Can we develop a model that fits multiple datasets and not just one specific dataset?

Detection technologies have matured over time, and greater depth and breadth of information is available for analysis, typically enriched with metadata and contextual information. Examples of such datasets include: attack events (honey net logs), network traces, and web crawler logs.

It is in the lack of support for identifying attacks across multiple data sources that we see the need for new research. This is because there is no framework to give us indicators for effectively combating cyber-attacks through combining information from various data sources, which limits the opportunities for a countermeasure development process. Therefore, more comprehensive correlation awareness needs to build upon this expanded information.

The high-level diagram of the proposed model for attack pattern recognition is given below: (see Figure. 1).

Figure 1: High level design of the proposed model for attack pattern recognition.

IV. APPROACH

One of the primary concepts behind our approach is the recognition of attack patterns, which represent a collection of related events, with the assumption that, for a given attack, two or more viewpoints will be available. Considerable efforts have been made to apply data mining techniques to problems related to network security, and these efforts have highlighted the use of these techniques to link analysis, neural networks and other machine learning approaches for intrusion detection. However, these efforts are focused on improvements to the intrusion detection system rather than finding evidence or discovery of new insights into analyzing complete phenomena of attacks using multiple datasets. Furthermore, only few common data mining techniques such as classification algorithms, association rules have been applied to raw network data but again it aimed at improving the alert classification to enhance the performance of the intrusion detection system. In our proposed approach we will take advantage of these available datasets that contain blueprints of attacks, hence our objective of using data mining is different from has approaches that have been used previously.

Moreover, to find unknown attack patterns we propose the use of unsupervised classification, as little prior knowledge about attacks is available. In addition, to reveal patterns that may result from combining evidence extracted from different sources, we will use an aggregation process that is highlighted as one of the modules of the proposed model.

The ResumeNet³ project provides a useful set of principles and design guidelines for our model. Resilience is the ability of a network to provide and maintain acceptable level of service in the face of various faults and challenges to normal operations [25]. ResumeNet uses a general two-phase high-level network resilience strategy, called $DR^2 + DR$: Defend, Detect, Remediate, Recover + Diagnose, Refine [26]. The first phase comprises the use of defensive (e.g. security) measures to protect the network from foreseeable challenges, such as cyber-attacks, detection of challenges in real time and subsequently
remediation of their effects before network operation is compromised, and finally recovery procedures. The second phase primarily involves improving service levels through the diagnosis and refinement of operations (see Figure 2).

![Resilience Strategy](image)

**Figure 2: Resilience Strategy**

Motivated by this high-level resilience strategy, the main components of our model are the following (see Figure 3):

- **Stage 1: Compute and Discover**: selection of attack features from a data set.
- **Stage 2: Extraction and Selection**: clustering, which aims to discover meaningful relations among patterns extracted from the dataset.
- **Stage 3: Aggregation and Refinement**: aggregation, i.e. the use of data fusion and updating the knowledge base of historic data profiles and patterns.

![Stages of the proposed model](image)

**Figure 3: Stages of the proposed model**

The stages of the model will be investigated using concrete examples such as the one in the next section.

### V. Attack Example

A Denial-of-Service (DoS) attack is an attempt to make a computer resource unavailable to its intended users. Typically, the targets are high-profile web servers. A Distributed Denial-of-Service (DDoS) attack occurs when multiple compromised systems flood the bandwidth or resources of a targeted system, again usually a web server. Attackers can use a variety of methods to compromise these systems, and some attacks may be multi-stage. For example, e-mail spam and malware are used first to gain control of several network nodes, then, a (D)DoS attack may be triggered to a specified target. We aim to use application-level information to detect attempted web server attacks. Traffic features, such as time, source IP address, and payload can then be used to review transport and network layer logs to check if other activities were occurring based on the same source address.

A simple way to visualize such a correlation is to imagine a series of events in the form of logs that are generated, each looking at different aspects of a network activity. After normalization, these logs can be placed on top of each other so that by looking across these layers one can identify significant commonalities in terms of the attack perspective.

### VI. Conclusion

In this position paper, we propose a framework for cyber situation awareness in computer networks. We believe it is possible to analyze multiple datasets with an expectation of gaining substantial insights into the diversity and types of the cyber challenges afflicting a network.

The lifetime of a cyber-attack can vary from days to months, and due to their nature, attributing different sources of events to the same attack is a challenging task. This is because the features of these attacks evolve over time, such as source or destination IP address, type of attack/exploit and so on. In addition, the fuzzy nature of attacks would make it difficult to model their behaviour.

A comparative study of clustering and classification techniques for finding security problems requires further research. Due to uncertainty and little prior knowledge of attack events, unsupervised classification techniques will be our focus.

### ACKNOWLEDGEMENTS

This research is partially supported by the EPSRC funded India-UK Advanced Technology Centre in Next Generation Networking.

### REFERENCES


Security Design Patterns in the MASTER Workbench

Paul J. Kearney, David A. Sinclair and Sebastian Wagner

Abstract—This paper describes pattern-related aspects of the prototype Protection and Assessment Workbench developed as part of the MASTER EU 7th Framework collaborative research project. The Workbench supports a model-driven design process within the overall MASTER methodology. It includes a Protection and Regulatory Model (PRM) tool that is a step towards turning the Workbench into an ‘organisational memory’ for design practices that accumulates and improves over time. PRMs are essentially control process design patterns that incorporate proven strategies in a re-usable form, saving time and improving quality and consistency.

Index Terms—business process, design pattern, model-driven design, security

I. INTRODUCTION

This paper will describe pattern-related aspects of the prototype Protection and Assessment (P&A) Workbench developed as part of the MASTER EU 7th Framework collaborative research project [1]. MASTER ran from 2008-2011 with the aim of providing a methodology and infrastructure that facilitate monitoring, enforcement, and auditing of security compliance, especially where highly dynamic service oriented architectures are used to support business process enactment in single, multi-domain, and iterated contexts.

The P&A Workbench is a prototype graphical software tool that supports a model-driven design process within the overall MASTER methodology. The term ‘Protection & Assessment’ reflects the focus of MASTER on design and execution of processes that enforce security policies and measure their effectiveness. The P&A Workbench is intended for use by an analyst/designer working in conjunction with a variety of business stakeholders to develop a Design Model. Typically, a Design Model is constructed, evaluated and elaborated iteratively until stakeholders agree that it describes an effective and affordable P&A solution that complies with control objectives. It is then refined further until the description is sufficiently concrete to provide the basis for implementation of a P&A system based on the MASTER infrastructure. Verification and Simulation tools are used to help confirm that the implemented system will indeed be fully compliant with the control objectives. The main output derived from the Design Model is a Policy Document that specifies rules that are interpreted by the run-time P&A system and determine its behaviour.

The Workbench comprises an extensible set of graphical modelling and transformation tools within a loosely coupled architectural framework, and a Repository in which interim and final models and other documents are stored. Collections of documents related to the same project are checked out of the Repository into a local workspace while being worked on using the Workbench tools. Documents in the Repository, notably Policy Documents, may be made available for deployment to, or consultation by the MASTER infrastructure components at run-time. Not strictly part of the Workbench, but closely related to it, are the MASTER Verification and Simulation tools. These are invoked from the workbench in order to test certain properties of the Design Model.

The Workbench achieved its objectives as a pre-competitive research prototype. It is a significant step towards the vision of a software tool to support a model-driven approach to the design and implementation of security controls and indicators derived from compliance requirements. The core functionality has been implemented, proving the concept and providing a basis for future development as a production tool. It is also suitable for trials and pilot projects for organisations considering adopting the MASTER approach. A production version would need to form part of an integrated suite or else inter-operate with leading modelling, development and management tools either as part of or mixed and matched on a best-of-breed basis. The prototype also provides an excellent platform for continued experimental implementation of advanced features, e.g. concerning knowledge management and re-use. Furthermore, the experience of developing the workbench has been useful in evaluating and building
Expertise in model-driven design via domain-specific languages (DSLs), and more specifically in use of the DSL-building tools from the Eclipse Modelling Project.

II. USAGE SCENARIO

A typical usage scenario for the workbench is as follows. A company already has a service-oriented infrastructure that it uses to partially automate the enactment of certain key business processes. It has identified that some high level security policies are relevant for the enactment of these business processes and the resources used. We are not concerned at this point whether the policies are due to regulatory requirements, industry codes of practice, or are of company-internal origin. The company now wishes to deploy IT controls to ensure as far as is practical that the policies are complied with and that the extent of any non-compliance is measured.

The company has decided to use the MASTER infrastructure to implement the controls. Company IT and security staff are now faced with a range of design decisions regarding how to use the ‘vanilla’ MASTER software to create an appropriate set of controls and indicators that interacts correctly with the existing infrastructure. The main means of customising the MASTER run-time infrastructure to implement the controls is to deploy a set of Signalling, Monitoring, Enforcement and Assessment policies to it.

Major requirements on the workbench are:

- to support the application of MASTER Methodology steps that guide the analyst and business stakeholders through description of the business context, selection and refinement of Control Objectives (COs) and Control Activities (CAs), specification of Key Assurance and Key Security Indicators (KAIs and KSI), and definition of Control Processes (CPs) that implement the CAs.
- to use the Verification and Simulation (V&S) tools to help confirm the correctness of the CPs, i.e. that composing the CPs with the relevant target business process will result in compliance with the COs.
- to facilitate implementation of these design decisions by automating the creation of MASTER Signalling, Monitoring, Enforcement and Assessment policies.

Furthermore, the company wants compliance to be auditable to provide its management, shareholders, customers, partners, and legal and regulatory authorities with confidence that the policies are indeed being followed. This means it must be easy to check that appropriate controls are deployed and that they have been implemented correctly. Here, ‘appropriate’ means that the controls are suitable to achieve the intention of the policies taking into account the anticipated threat environment and the risk appetite and budget of the organisation. The workbench therefore needs, in addition, to enable an independent auditor to review and verify design and deployment decisions to ensure that appropriate choices have been made to implement the high level policies.

The company will need to update the controls as requirements and business, technology and threat environments change. Furthermore, it is unlikely that the company will implement controls for a single process in isolation (except as a pilot), but rather would roll MASTER out for all relevant business processes as part of a major compliance or Business Process Re-engineering initiative. The Workbench therefore needs to support the maintenance of controls over time and ideally, re-use of successful controls and accumulation of experience.

III. SECURITY PATTERNS

Schumacher et al. [2], inspired by the approach to pattern-oriented software architecture taken in [3] give the following definition: “A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular structure.” Security problems typically deal with constraining the behaviour of systems to uphold confidentiality and integrity properties while continuing to provide a service to legitimate users, in the face of malicious or misguided agents. A security pattern establishes an approach to solving a class of security problems by generalising over related successful cases.

Schumacher et al. regard humans as being the only audience for patterns. They further say that it is not practical to make patterns machine-readable and automatable, as it is all but impossible to formalise them. They contrast patterns with “other design or modelling techniques such as the Unified Modelling Language” that result in artefacts that are intended to be readable by machines and humans. This view is not universal, however; for. For example, Sowa identifies three kinds of pattern (syntax, semantics and pragmatics) and three kinds of notation (natural language, linear notations for logic and computation, and graphical diagrams and movies) in his work applying conceptual graphs to knowledge design patterns [4].

The approach taken in MASTER attempts to bridge the two worlds of patterns and modelling languages. A MASTER pattern (termed a Protection & Regulatory Model, PRM) includes fields for textual / free form entries that are aimed purely at a human audience and are broadly similar to those found in ‘classical’ patterns. However, it also contains corresponding design fragments expressed in the MASTER modelling language.

The intention is that the Workbench will incorporate a library of PRMs. In the course of developing a design model, the analyst and stakeholders will browse this library looking for PRMs that are relevant to the application context and the aspect of the model being worked on. For example, the analyst may be working on a control objective aimed at minimising opportunities for insider fraud and find a pattern describing...
how to do this using the principle of separation of duties. This stage is similar to searching a conventional pattern system.

The analyst then turns to the model fragments described in the PRM. These are templates with place-holders that must eventually be unified with entities in the model. The template is instantiated and added to model, and the analyst stitches it into place by associating existing entities with placeholders. Often, the existing model will need to be re-worked to enable a good fit, for example, a role in a business process might have to be divided into two simpler roles to allow separation of duties. Sometimes, the changes required to achieve a fit may be judged excessive, and the pattern will be discarded and a new one sought. Not all the place-holders need to match existing model elements; indeed some may give rise to further searches of the pattern library.

Use of patterns in this way results in improved quality (as a result of adopting proven solutions and best-practice) and also in increased productivity as models are constructed partly from pre-fabricated components rather than always working at the level of the primitives of the modelling language.

Once a project is completed, the model is reviewed with the aim of identifying successful strategies with potential for reuse. These are recast in more abstract form, documented as PRMs and deposited in the library. PRMs that have been used should also be critiqued as part of this review and comments added to the record in the library. If appropriate, a PRM may be modified or elaborated in the light of experience, or even deleted if it has proved unsuccessful on several occasions. In this way, the Workbench will become an ‘organisational memory’ for design practices that accumulates and improves over time.

PRMs are now covered in more detail, before moving on to describe an early prototype PRM tool that was implemented as part of Workbench during the MASTER project.

**IV. PROTECTION AND REGULATORY MODELS**

Each PRM describes a design pattern that captures “best practice” in designing control processes for a specific control objective in a specific context. The essence of a PRM is a set of parameterised fragments of control processes, and each PRM is linked to a generalised control objective. The control process fragments in each PRM are composed and sequenced in a specific manner. Each control process fragment is defined by a set of allowed event traces. An event trace is a sequence of observable events. These event traces may consume events from or add events to its enclosing environment. Some of the events in the traces may be bound to terms in the PRM’s control objective or the events that are consumed or added to its enclosing environment. These events are defined as exposed PRM events and are included in the parameters of the PRM. Non-exposed events, internal PRM events, are not visible to the actors who interact with the PRM. Internal PRM events can be renamed without any impact on actors and processes that interact with the PRM. Exposed PRM events are typically the input and output events of the PRM, but may also include intermediate events.

Each PRM contains the following elements.

1) A textual description that describes the purposes of the PRM, its associated control objective(s) and the type of environments in which the PRM is envisaged to be deployed. This description should be brief enough to allow a user to browse this description within a collection of PRMs, yet it should contain sufficient information to allow a user to make a preliminary decision on whether a PRM is appropriate for a particular design.

2) A description of the parameters of the PRM.

3) A description of the context in which the problem that is being solved arises. In addition to describing the attacks that the PRM is addressing, the section also describes the organisational context in which the PRM can be used. More specifically the PRM describes the requirements and restrictions on the organisational structure in which the PRM can be used in terms of:

   a) The entities participating in the solution: Actor roles, Goals, Activities, and Resources.

   b) The structural dependencies among entities. These structural dependencies can arise due to the context of the problem the PRM is addressing or due to the structure of an organisation. An organisation can impose dependencies on entities and actors. For example, the organisation may require certain entities collaborate together. It may define which actors are subordinate to other actors, and which actors have incompatible roles and hence require a separation of their duties.

4) A description of any infrastructural services required by the PRM. For example, an authentication PRM may require a service that associates a set of roles with each user.

5) A description of the control objective implemented by the PRM. This is given by a textual description and a formal statement of the control objective using the MASTER Property Specification Language.

6) A formal description of the control process fragments that implement the PRM. This description is specified in the M-calculus, which has both a graphical and textual representation. The M-calculus is a process calculus that extends Milner’s π-calculus [5] to include timed events and probabilistic events. The motivation for describing the control process fragments formally is that this enables the PRM designer to prove that the PRM implements its control objective in the specified contexts.

7) A description of the possible impact of the PRM on the overall system. This section should address, but not be limited to, issues such as availability, confidentiality, integrity, performance and usability.

When composing multiple PRMs, care needs to be taken that the names of the exposed PRM events do not conflict. Internal PRM events can be renamed to avoid any conflict with events in the target processes, control processes or other
PRMs. Since the exposed PRM events are all parameters of the PRM, each PRM is instantiated with events from the target and control processes. If two PRMs reference the same event from the target and/or control processes, this event is called a conflicted event. Conflicted events may require special handling, such as duplication to ensure that each PRM can consume the event, or amalgamation to ensure that conflicted events produced by both PRMs do not invalidate the system’s control objectives when consumed by the environment.

Since PRMs represent “best practice” they have an added significance in environment with multiple trust domains. While organisation X may be reluctant to disclose the details of the control processes it is providing to organisation Y, if organisation X agrees to provide a specific authentication PRM then organisation Y can be sure of that the corresponding control objective is achieved (subject to the PRM’s assumptions on its environment) without organisation Y revealing, or allowing access, to the control infrastructure within organisation Y.

The use of PRMs can be illustrated by looking at how one could be used to represent a control taken from ISO 17779-2005. The ISO standard describes Control 11.1.1 “Access Control Policy” as “An access control policy should be established, documented and reviewed based on the business and security requirement for access”. The text does not describe how to implement Control 11.1.1, but it does describe some properties that any implementation of the control 11.1.1 must have. The corresponding PRM would capture a “best practice” implementation of this control in specific contexts and outline how the properties specified for this control can be proved in these contexts. The main objective of PRM is to capture precisely and formalise the knowledge required to implement a control process aiming to be compliant with control objective derived from a business objective or regulation. The control objectives describe “patterns” that compliant control processes must match and PRMs formalise these patterns.

V. PRM TOOL

The prototype PRM tool is an Eclipse plugin that is a step towards turning the Workbench into an ‘organisational memory’ for design practices that accumulates and improves over time. As mentioned in the previous section, PRMs are essentially control process design patterns that incorporate proven strategies in a re-usable form, saving time and improving quality and consistency. They map security compliance objectives to control processes and architectures that achieve them in defined business and technical contexts. The tool allows new PRMs to be created and stored in the Repository, and retrieved and instantiated as part of a new model when required.

During its initial creation the PRM elements and properties described in Section Error! Reference source not found. (name, description, problem context, parameters etc.) can be defined with the PRM tool. The solution of a PRM including its control objectives and the implementation artefacts, i.e. control processes and the corresponding infrastructure services, can be directly linked to elements modelled within the design workbench. Hence, if they are modified with their corresponding workbench tools (e.g. a process model editor) these modifications are immediately visible within the PRM. Moreover, this has the advantage that these elements can be also used in other PRMs. The PRM in Figure 1 describes a Separation of Duty (SoD) control pattern that ensures that two roles R1 and R2 performing critical operations within a business process BP are played by two separate agents. The PRM is implemented by a corresponding control process that enforces that the control objective is met. The control process depends on the infrastructural service AUTHZ that acts as authorization manager, and PROC that holds the descriptions of the process instances of BP the control process is applied to. The services and the control process for this PRM are defined in the design model SoD_Fragments and can also be used by other PRMs. The business process and the roles the control process is applied to are use-case dependent. Hence, these elements are parameterized. To achieve that, parameters can be defined on a PRM with the PRM tool. In our scenario there are three parameters: BP, R1 and R2. Each parameter represents an abstract element to be replaced by a concrete element that provides domain-specific information during the instantiation of a PRM in order to make the implementation artefacts executable. For instance the parameter BP in Figure 1 refers to an abstract Process element in the design model SoD_Fragments. During instantiation of the SoD PRM the abstract Process element and the Role elements have to be replaced by a concrete implementation these elements.

The PRM tool provides a wizard that guides the user through the instantiation of a PRM. As shown in Figure 2, the
wizard lists all parameters of a PRM and the user can provide concrete values for the listed parameters. More precisely, this means that she can supply concrete design model elements from arbitrary design models from the workbench or repository, respectively. In order to do that for each parameter she provides the path to the design model that contains the element that has to serve as parameter value and selects the design model element from the combo box. Our SoD PRM has to be applied to a loan approval process. Hence, as shown in Figure 2 the BP parameter is instantiated with a Credit Process. The combo boxes list only elements of a design model that match the type of the parameter defined by the abstract element. Therefore, to instantiate the first role R1, only those elements of the selected design model that have the type Role are listed in the combo box. Figure 2 lists, in this case Auditor, Clerk and Manager.

![Figure 2 PRM Instantiation Wizard](image)

After the user has provided the values for each parameter she finishes the wizard and the PRM is instantiated by creating a new design model out of it that is stored in the workbench. In our example a design model is created that contains the Credit Process and the SoD control process that can be performed for the specified roles along with the infrastructure services.

VI. FUTURE DIRECTIONS

The current prototype provides an excellent platform for continued experimental implementation of aspects of the Workbench vision that we did not have time to investigate or implement fully during the MASTER project. Re-use of parameterised models in design patterns is already present in the workbench in the form of the PRM tool. While this is effective in proving and demonstrating a viable approach there are many details to be fleshed out. Once a framework for re-use has been established, PRM libraries need to be developed providing abstract solutions to compliance requirement commonly appearing in regulations and corporate policies.

PRMs are only one form of knowledge management that would be useful in a Workbench. The design process makes extensive use of knowledge derived from experience. Traditionally, this has been held in the heads of experts, and in text books, codes of practice and standards. A future research project could incorporate an expert system into the Workbench to provide advice on design decisions. The knowledge base of the expert system could be seeded with generic guidance, and expanded over time with enterprise-specific and sector-specific knowledge to become a living means of continuous improvement. An important aspect of the project would be investigation of means of capturing new knowledge and otherwise maintaining the knowledge base.

REFERENCES

Extending AOP Principles for the Description of Network Security Patterns

David Llewellyn-Jones, Qi Shi, Madjid Merabti
School of Computing and Mathematical Sciences
Liverpool John Moores University
Liverpool, UK
{D.Llewellyn-Jones, Q.Shi, M.Merabti}@ljmu.ac.uk

Abstract—Aspect Oriented Programming is increasingly being used for the practical coding of cross-cutting concerns woven throughout an application. However, most existing AOP point-cut definition languages don't distinguish in their application between different systems across a network. For network security there is a need to apply different aspects depending on the role a piece of code has within the larger networked system, and a new approach for this is therefore required. In this paper we present a formalism for how this might be approached, proposing a way to capture distributed point-cuts for applying different aspects in different parts of the network. The method is based on templates that match properties within the code, and a set of flexible relationships that can be defined between them.

I. INTRODUCTION

While the majority of modern, mainstream programming languages have adopted object-oriented programming (OOP) approaches, it's nonetheless widely acknowledged that the OOP paradigm fails to adequately reflect the full breadth of structural features applicable to software development. In particular, while objects are a great way to compartmentalise functionality into reusable components, they can make it difficult to introduce functionality spanning multiple components. Aspect-oriented programming (AOP) has become established as an effective way for addressing this.

Although AOP can be achieved statically at compile-time, the most successful AOP technologies have drawn upon the reflective (introspection and intercession) capabilities of modern languages (e.g., those built using virtual machine abstractions, such as Java, .Net, etc.) to provide dynamic aspects that can be introduced and removed at run-time.

From a security perspective, AOP techniques are an exciting development that offer considerable promise. Along with logging, redundancy and Quality of Service (QoS), security features are invariably considered as canonical examples of cross-cutting concerns particularly amenable to AOP approaches. However, security concerns present particular challenges, not least because the most pressing and relevant cases occur when components are distributed across a network. In contrast, most AOP approaches still focus on individual pieces of software, with aspects woven into the code in multiple places, but without considering how concerns cut across multiple interacting networked systems.

While security features should therefore be ideal AOP candidates, the need to introduce them across distributed networked systems creates both conceptual and practical challenges. For example, how can we describe concerns to allow reasoning across multiple heterogeneous components? Having established a suitably expressive language, how can aspects be applied simultaneously and so as not to inadvertently introduce new security vulnerabilities?

In this paper we focus on the question of describing suitable patterns. While a number of AOP methodologies have been developed with distributed systems in mind, including several that consider security in particular, we believe there is still scope for improvements in terms of their expressiveness, and that these improvements are needed to allow the wider variety of security requirements found in networked systems to be adequately described.

The remainder of this paper is structured as follows. In the next section we will consider a selection of the existing proposed AOP approaches applicable to distributed systems. We build on this in Section III by considering examples issues that these systems do not yet address, but which are needed for security patterns in networked systems. Section IV turns these into a set of requirements we aim to fulfil, with the resulting point-cut definition language described in Section V, followed by applications in Section VI. Finally we conclude and suggest future work in Section VII.

II. RELATED WORK

Broadly speaking, AOP involves the introduction of aspects into a piece of software. These aspects represent cross-cutting concerns, so that the same (or similar) functionality can be added in the same way across multiple components (e.g., objects or methods) within the same piece of software.

There are already a number of well-established AOP platforms, including AspectJ, PROSE, JAsCo, LOOM.Net and DSAW. One of the important characteristics that distinguishes these different platforms is the selection of join-points each supports. A join-point represents the possible types of location where aspects can be introduced into an existing body of code. Common examples of join-points...
include before, after or around a method call (where the latter potentially allows the call to be replaced entirely). In addition, some platforms will allow new attributes to be added to objects, or alteration of attributes through changes to the relevant getter or setter methods. Other language features, such as operator overloading, ensure even a relatively restricted set of join-points can nonetheless offer powerful and flexible program extensions.

Join-points provide a set of potentials which are turned into actuals through the use of point-cuts and advice. A point-cut is a description of a precise set of join-points, restricted for example by method name (often using wildcards) or parameter types. Each is associated with an advice, which constitutes the code to be injected at each of the join-points satisfying the point-cut description.

These existing techniques are well-suited to individual pieces of software, but require extension in order to be applied in distributed environments. For example, Pawlak et al. [1] developed A-TOS, a “general purpose aspect-oriented reflective middleware” to introduce such extensions. In A-TOS aspects are encapsulated in aspect components, which describe both wrappers that are applied to – and respond to events from – the existing distributed objects, and centralised code that provides a global state and semantics available to all the individual wrappers.

This provides a clean way for describing distributed aspects, and the authors also apply it to security in particular, demonstrating how Kerberos-like authentication can be achieved using the technique. However, while the system utilises the network to offer a global state to the aspect components, it provides limited means for reasoning about how the wrappers should be applied (e.g. refining the choice of join-points based on the nature of the components).

A different approach is taken by Duzan et al. [2], by focussing on the boundary between a component and the middleware used to abstract the network functionality (e.g. CORBA). This work allows advice to be added across client-server connections, for example to allow different data to be sent from the server depending on the network conditions at the client end. The work introduces the concept of regions, which allow different code to be executed depending on dynamic properties such as QoS parameters defined using a contract. The work focusses tightly on QoS, but the use of contracts to define dynamic behaviour dependent on external conditions could potentially be adapted for security properties.

The DiaAspect Architecture Description Language [3] also focusses on distributed middleware, but in this case as an extension of existing modelling languages such as UML. The DiaAspect models can be translated into more specific implementations based on technologies such as RMI or Web Services. A carefully tailored join-point model allows point-cuts to be defined based on a variety of messaging patterns that include commands, publish/subscribe events, network sessions, component registration/discovery, and so on. Advice is injected at both the message emitter and receiver ends. As a modelling language DiaAspect is potentially well-suited to the abstraction of patterns, and the authors demonstrate its use for both SSL certificate management and access control.

At the implementation end, Horie et al. [4] take synchronisation of advice injection as a cross-cutting concern itself. They show how aspects can be used to bootstrap the synchronisation of other aspects by first injecting code to separate deployment from activation, before using this to simultaneously activate further aspects. A method is proposed for dynamically introducing encryption/decryption that avoids packet loss that would occur if the encryption code were to be activated before the decryption code.

III. AOP SECURITY CHALLENGES

While all of these techniques tackle important issues related to applying distributed aspects, one area which is left open is that of reasoning about how aspects should be applied across multiple distributed components. Invariably a binary client/server-style relationship is assumed. While in many cases aspects can be easily introduced across connections (i.e. different coordinated code at the client and server ends of a connection) there is no coordination across multiple components.

We therefore explore how patterns can be defined to allow aspects to be woven into multiple components straddling multiple network connections. To understand the requirements of this approach we first consider a number of example scenarios.

A. Secure Data Forwarding

A very simple example of a security requirement is where data should be accessible by one system but not another (e.g. as a result of data classified at different security levels, as shown in Figure 1.a). This can be enforced using AOP at network endpoints, by designing aspects to add metadata and encryption at one end of a network connection, then check
the metadata and decrypt it at the other end. This scenario can be captured using aspects applied to pairs of atomic services and can already be represented using existing (e.g. client-server) methods.

However, this scenario can be extended, where for example data might be allowed to travel through a system not entitled to access it, as long as decryption doesn’t take place at the intermediate systems (See Figure 1.b). This requires a more complex pattern description than a simple binary relation.

B. End-to-end Security

Extending the example in Section III-A, we might consider a case where endpoint security must be applied using aspects at the source and destination node, but tackled differently at intermediate nodes (see Figure 2).

In this case, a pattern capturing the chain of nodes, but where the length of the chain is indeterminate, is needed. Current methods have difficulty capturing this case adequately.

C. Separation of Duty

Not all patterns involve linear chains and it may be necessary to match different branch-types within a network graph. For example, a separation of duty requirement might allow component A in Figure 3 to access B only if it had not yet accessed C (or vice versa). Aspects could be woven into A, B and C to enforce this, but would need to be applied only where such a split occurs.

IV. REQUIREMENTS

In order to allow the above concepts to be formalised through pattern descriptions, we formulate a number of requirements that we believe a distributed point-cut definition language must fulfill.

1) Patterns must capture sequences of more than two systems.
2) Patterns may match multiple (potentially infinite) sets of networked system. Flexibility is needed in both the depth (length of sequences) and breadth (number of links entering/leaving a system) in the network.
3) Patterns must not be indeterminate. That is, while there may be multiple arrangements that satisfy a pattern, the question of whether a particular pattern matches a particular set of systems should be decidable.
4) Pattern matches should be based on both the contents of the system code (obtained through introspection) and the relationship between systems (the structure).
5) It should be possible to relate the aspect code with the distributed point-cut and associated code. For example a different aspect is needed at either end of an asymmetric encrypt-decrypt network link.

V. POINT-CUT DEFINITION LANGUAGE

In order to fulfil the above requirements we define a formalism for specifying patterns that allows system properties to be combined with network interactions. Our intention is to allow a language to be developed from this for the flexible definition of distributed AOP point-cuts and advice.

In essence, we aim to specify a graph, where each node in the graph represents an atomic system (a piece of code to which an aspect can be applied), and the links between nodes represents possible interactions between code across the network.

However, we must generalise this, so that the nodes represent sets of atomic systems (defined by a set of properties that a system must match). Consequently we distinguish between an actual graph of interacting systems and a template graph. The actual graph represents a set of networked systems. If each atomic system (nodes in the actual graph) fits the properties of a respective node in the template graph, and if the interactions also match, then the template matches the actual. Figure 4.a shows an example of a template, with various properties $p_1, \ldots, p_5$, and an actual graph that might match it in Figure 4.b with actual systems $n_1, \ldots, n_6$.

This would allow us to test templates against actual graphs, but doesn’t tell us how to apply any aspects. For each node in the template graph we therefore assign a point-cut template and advice to apply to that system. These are
represented by the point-cuts and advice \(a_1, \ldots, a_4\) in Figure 4.a.

Unfortunately such a system is still too restrictive. In many cases we want to apply advice to networked systems without being so prescriptive about the interactions between them. For example, in a client-server setting, we may want to specify advice to apply to the client and server separately.

We can capture this with the graph shown in Figure 5.a (where \(C\) represents client code and \(S\) represents server code), but this will then fail to match if the server is managing multiple connections as in Figure 5.b. Flexibility is therefore needed to define which connections are relevant for a template, which can be ignored and which are prohibited (i.e. result in a template that doesn’t match).

In order to fulfil Requirement 2 we must therefore allow flexibility in the way the input and output patterns are defined. Presenting this diagrammatically is difficult: how should we represent a set of – potentially infinite – distinct input and output patterns on a single diagram? We therefore consider a more formal description.

First we establish some definitions. Let \(N\) be the set of actual services in the scope of consideration, with \(n_1, \ldots, n_j \in N\) being actual systems as above. Let \(P\) be a set of property descriptions and \(A\) be the set of point-cut advice definitions with \(p_1, \ldots, p_k \in P\) and \(a_1, \ldots, a_l \in A\). The \(a_i\) can be written in any existing AOP point-cut specification language such as that provided by DSAW [5]. We leave the choice of language for both the property definition and aspects open at this stage.

Let \(S\) be a set of states; for simplicity we can assume \(S = \mathbb{N}\) (the natural numbers) using some kind of encoding. In practice it may be convenient to give \(S\) a more defined structure.

It will be convenient for us to define sets of properties, as well as sets of properties with associated actions (advise and point-cuts). We therefore note that these can be defined as any member from the sets \(2^P\) and \(2^{P \times A}\) respectively, where \(2^X\) represents the power set of \(X\). A member \(x \in 2^P\) can be any subset of \(P\), so takes the form

\[x = \{p_1, \ldots, p_j\}\]

for some \(j \in N\) and \(p_1 \in P\). A member \(y \in 2^{P \times A}\) takes the form

\[y = \{(p_1, a_1), \ldots, (p_k, a_k)\}\]

for some \(k \in \mathbb{N}\), \(p_i \in P\) and \(a_i \in A\).

For a given node satisfying a given set of properties \(p \in P\) we want to define a set of possible templates to match it against. A single template is defined as follows.

\[(a, x_1, x_O, \bar{x}_I, \bar{x}_O) \in A \times 2^{P \times A} \times 2^{P \times A} \times 2^P \times 2^P\]

We call this an **atomic template** since it allows a specific set of input and output properties to be specified as a template for a single atomic system. We set \(T\) to be the set of all atomic templates:

\[T = A \times 2^{P \times A} \times 2^{P \times A} \times 2^P \times 2^P\]

Suppose we have a node \(n\) satisfying the properties of \(p\). Then in the atomic template for \(p\) the set \(x_1\) represents the set of properties that any node with inputs to \(n\) must satisfy; \(x_O\) represents the set of properties that must be satisfied by any node \(n\) connecting to; \(\bar{x}_I\) represents the set of properties that any node with inputs to \(n\) must not satisfy; \(\bar{x}_O\) represents the set of properties that any node \(n\) connects to must not satisfy. Note also that each property in \(x_1\) and \(x_O\) has an advice associated with it. In case the complete template holds, this advice will be applied to the related actual node satisfying the property, which we will discuss in more detail shortly.

The above defines a template for only one property. We must extend this to allow templates for graphs of properties (that will then be matched against actual systems). We do this by specifying a function \(f\) of the following form.

\[f : P \times S \rightarrow 2^{T \times S}\]

The function \(f\) defines a single general template and can be used to match against sets of actual systems. Note that as well as the atomic template, the function also maps to and from an input and output state \(S\). This is to allow the same atomic template to apply at different places in the graph in different ways. We will assume that the initial state \(s_1\) for any general template must be zero.

So for example, consider the arrangement from Figure 4.b again. The system \(n_1\) satisfies the property \(p_1\), hence we first apply the function to \((p_1, 0)\). The function will return a set of associated atomic templates \(t_1, \ldots, t_j\) and an updated state \(s_2\). We can then test the inputs and outputs of \(n_1\) against the properties in each of the \(t_i\) to establish whether any of these hold. If they do, the process is repeated recursively, applying the function to each of the atomic systems that \(n_1\) connects to/from, along with the updated state \(s_j\). Note that at any point it may be possible for multiple of the \(t_i\) to match. In this case, each is tested separately. The process completes if either every recursive branch fails to match at some level, or one of the templates matches in full.
In the event that the template matches in full, the aspects defined in the template should then be applied to their respective systems.

To understand this better, in the next section we will return to our examples from Section III to see how such functions can be defined.

VI. APPLICATION

In order to demonstrate further the application of the pattern descriptions, we consider how they might apply to the examples from Section III. For secure data forwarding the key is to apply four different aspects at different points across the sequence of systems. We can do this by defining the function \( f \) to contain the following mappings.

\[
\begin{align*}
(H, 0) &\rightarrow (\emptyset, \emptyset, \{(L, a^H_I), \emptyset, \emptyset, 0) , \\
(L, 0) &\rightarrow (\emptyset, \{(H, a^L_I), \{(L, a^L_O), \emptyset, \emptyset, 0) , \\
(H, 0) &\rightarrow (\emptyset, \{(L, a^H_O), \emptyset, \emptyset, 0) ,
\end{align*}
\]

with all other inputs mapping to the empty atomic template. This matches only sequences of the form \( H \rightarrow L \rightarrow H \).

Note that for the above template any other inputs or outputs are ignored, but we could set it to fail if there are any other inputs or outputs by setting the \( x_I \) and \( x_O \) sets accordingly.

For the case of end-to-end security we require a slightly more flexible function. In this case we define the following mappings (with all others mapping to the empty template).

\[
\begin{align*}
(A, 0) &\rightarrow (\emptyset, \emptyset, \{(B, a^A_I), \emptyset, \emptyset, 1) , \\
(B, 1) &\rightarrow (\emptyset, \{(X_1, a^X_I), \{(X_2, a^X_O), \emptyset, \emptyset, 1) , \\
(A, 1) &\rightarrow (\emptyset, \{(B, a^A_O), \emptyset, \emptyset, 2) ,
\end{align*}
\]

where \( X_1, X_2 \) are one of \( A, B \) and \( Y_1, Y_2 \) are one of \( B, A \) respectively. This generates a total of six templates. The first requires a sequence \( A \rightarrow B \). We then have four mappings that allow any sequence of the form \( A \rightarrow B \rightarrow B \rightarrow B \rightarrow A \rightarrow A \). Note that the last of these could be left out if we required at least one connection through \( B \). The last template stipulates the input requirement of the final \( A \) node. This is needed in order to ensure the sequence of templates matches through to the end.

Finally, the separation of duty requirement could be specified using a function \( f \) with the following mappings.

\[
\begin{align*}
(A, 0) &\rightarrow (\emptyset, \emptyset, \{(B, a^A_I), \{(C, a^C_O), \emptyset, \emptyset, 1) , \\
(X, 1) &\rightarrow (\emptyset, \{(A, a^X_I), \emptyset, \emptyset, 1) ,
\end{align*}
\]

where \( X \in \{B, C\} \), generating three rules in total.

This case is slightly different, since we have two tuples in the \( x_O \) set of \( A \): one for \( B \) and another for \( C \). However, note that the rules for \( B \) and \( C \) have only a single input, since the template only sees the limited view of the world from the perspective of \( B \) or \( C \) respectively. Any interactions that requiring a larger world-view must capture this through the state variable.

VII. CONCLUSION

In this paper we have briefly outlined a formalism for applying aspects to networked systems which extends existing techniques in a novel way, thereby providing greater flexibility. The approach allows complex structures to be defined across multiple networked systems using mappings between properties and atomic templates to which actual systems can be matched. While this provides the high-level mechanisms, there are a number of significant gaps in the work. For example, we did not present the point-cut language or explain how properties can be specified. Although an existing point-cut and advice language could be used, a more complete solution would also integrate state values into the advice.

Our approach as presented here is also theoretical, and doesn’t explain how a template function would be specified in practice. All of the examples given here require only a finite set of possibilities for the inputs and outputs from any given system, which allows us to specify them as a finite list of atomic templates. The reason we use a function rather than a list is to allow more flexibility than this, and in general it wouldn’t be possible to represent \( f \) using a finite list. In our future work, we therefore aim to demonstrate how these issues can be overcome in practice, using a practical implementation for applying distributed AOP security techniques.

REFERENCES


On the use of Design Patterns to Capture Memory Corruption Vulnerabilities

Nick Walker, Natalie Coull, Ian Ferguson, Allan Milne
School of Engineering, Computing and Applied Maths
University of Abertay Dundee
Bell Street, Dundee, DD1 1HG

Abstract - Computer systems are continuously subjected to cyber-attacks seeking to breach existing security measures. Software development companies conduct testing of their code prior to releasing their product, yet exploitable vulnerabilities are still prevalent in many applications. Memory corruption vulnerabilities are amongst the most difficult to detect, but can be the most damaging if exploited. This research presents both an effective taxonomy of these vulnerabilities, which can be used to identify software threats and a methodology to maximise the number of memory corruption vulnerabilities that are identified during software testing. A means of cataloguing such vulnerabilities was required: As design patterns were already familiar to software engineers the use of a pattern language seemed appropriate, particularly as the solution to the vulnerabilities lay in the software engineering domain.

Keywords memory corruption vulnerabilities, taxonomy, methodology, design patterns

I. INTRODUCTION

By far the most diverse and prevalent type of software bugs, with the greatest security implications are memory corruption vulnerabilities. A survey undertaken by Chen, Xu and Nakka [1] indicates that memory corruption vulnerabilities account for 67% of all software flaws identified. These memory corruption vulnerabilities can manifest themselves in various forms and occur when the contents of a memory location are modified unintentionally or maliciously and result in the corruption of data, through logic or programming errors. These vulnerabilities can exist at a number of different stages in program execution [2], making it extremely challenging to locate them during normal software testing procedures. This is increasingly difficult if the tester has limited understanding of them and how to find them. A further difficulty is encountered when analysing the effects of memory corruption in order to identify the source of a given vulnerability. The source of a memory corruption vulnerability and how it manifests in terms of the program crash may be far removed in terms of covered program code, with tedious links between source and manifestation.

If true security is to be achieved, programmatical issues must be identified and countermeasures applied proactively by fixing the erroneous code, rather than trying to reactively prevent exploitation of the system. To achieve this requires considerable understanding of the different types of memory corruption vulnerabilities and a well-planned systematic approach to identify the existence of these errors must be undertaken.

The use of patterns to catalogue idiom from the security field is an appropriate way of presenting such ideas to practising software engineers. As the type of vulnerabilities under consideration (memory corruption) is largely created by ignorance during the software development process, anything that could be done to communicate an awareness of both the problem and of appropriate solutions would be beneficial.

II. LITERATURE REVIEW

Whilst a full review of the security patterns literature is beyond the scope of this paper, (which will limit itself to an overview of the area and a selection of work related to the current topic) useful surveys can be found by Yoshioka et al.[3], Hafiz et al.[4] and more recently Uzunov et al.[5]

Design patterns have their origins in the field of architecture(Alexander[6]), were adopted for use in software engineering by Beck and Cunningham[7] and have been popularised by the “Gang of Four” book[8] by Gamma et al. Their adoption in the area of security is generally credited to Yoder and Barcalow.[9] Various attempts at cataloguing security patterns have been made including those of Yoshioka et al.[3], Schumacher et al.[10], Steel et al.[11], Yskout et al.[12] and Scandariato et al.[13], Dougherty et al.[14] and Kienzle et al.[15]. The problem of classifying security patterns (or the production of taxonomies) once they have been catalogued has been tackled by (amongst others) Hafiz[4], Fernandez et al.[16], Washizaki et al.[17], Van Hilst et al.[18], Heyman et al.[19] and Bunke et al.[20].

Kis[21] proposed the use of antipatterns to describe security vulnerabilities – an approach which seemed appropriate for our work based as it is on the discovery and cataloguing of new vulnerabilities. An antipattern, however, documents how not to do something. Whilst this may appear simply a matter of perspective, the authors wished to enable the cataloguing of solutions (i.e. how to address the vulnerability), and as such it is argued that the description of a vulnerability is a component of the description of the problem and hence this work is not considered to be in antipatterns.
The work on cataloguing attack patterns (specifically that of Wiesauer and Sametinger[22]) is most closely related to the problem under consideration, but whilst an attack may exploit a vulnerability, it is not, in itself, a vulnerability. There thus exists a qualitative difference between the “attack pattern” and the “vulnerability pattern” approaches. It could be argued that the cataloguing of vulnerabilities is too specific to require a pattern approach, however support for the suitability of the pattern metaphor comes from two sources: a) The current work can be seen as an extension of the taxonomy of Wiesauer and Sametinger[22] and b) The description of a vulnerability fits naturally into the problem field of a pattern template.

Other significant sources of influence on the work were the various attempts made to catalogue vulnerabilities that have become apparent as a consequence of successful attacks (rather than explicitly search for them) including the CAPEC[23], CVE[24] and NVB[25] databases.

III. SUMMARY OF WORK

In order to reduce the number of memory corruption vulnerabilities that are present in code, software engineers and testers need to be able to identify aspects of the code that could be exploited. Our taxonomy of these vulnerabilities could be used as a checklist to ensure that opportunities for exploiting code is minimized. Following an evaluation of existing literature and taxonomies on memory corruption vulnerabilities, we have identified the following criteria on which the bugs should be classified. These are:

1. Origin of vulnerability
2. Behaviour of vulnerability
3. Criteria required to leverage vulnerability
4. Impact of vulnerability
5. Countermeasures required to solve the vulnerability

These criteria are the “fields” of the ad-hoc pattern language. The taxonomy categorises the bugs based on simple behavioural facts about the target code, rather than relying on the highly variable contextual criteria that are used by other taxonomies. This behaviour-based approach proves very effective in determining the existence of software vulnerabilities, as the mechanisms that introduce issues into a code base are polymorphic in nature. This means that slight changes at a source code level can have cascading effects on the compiled instance of the code, making successful categorisation based on either source code or decompilation techniques improbable. The focus on behavioural criteria gives the taxonomy its strength when used to determine the existence of a software issue. It should be noted that this taxonomy is unique in criteria 3 and 5, which would be invaluable for a software tester to determine the impact of the vulnerability and how to solve it.

To identify the range of memory corruption vulnerabilities that should be included in the taxonomy, we conducted a thorough evaluation of a range of sources (for example CAPEC [23], CVE [24], OWASP [26]), which include descriptions of bugs that allow the contents of memory location to be modified. This review identified those bugs where corruption was possible and established their defining behaviour and effects. Any bugs which overlapped, for example they had the same defining origin (field 1 of our taxonomy), were combined as they are different manifestations of the same core bug. The complete taxonomy consists of nine different types of memory corruption bugs: Strcpy(), Strncpy() and Memcpy() overflows, Free of Non Dynamic Memory, Free After Malloc() Failure, Double Free, Array out of Bounds, Pointer Reassignment to Double Free and Format String vulnerabilities. Table 1 shows the memory corruption vulnerabilities and describes them according to our taxonomy. While it is not feasible to discuss each corruption bug in the taxonomy, the Double Free() memory corruption vulnerability is discussed as an example.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[])
{
    char *p = malloc(128*sizeof(char));
    free(p);
    free(p);
    return 0;
}
```

Fig 1 Example of the Double Free Vulnerability

The code in Fig 1 is an oversimplified example of a ‘double free’ vulnerability. It occurs when a pointer that is valid and has been allocated using malloc() is freed, and then later is erroneously freed again without ever being reallocated. At the first free, no corruption occurs as this is correct dynamic memory management, however when the second call is made, the linked list values that control what is stored on the heap have already had the pointer *p removed, and therefore the second call corrupts the heap structures by freeing a section of memory that is unexpected (as the memory may have been reallocated for different purposes since the first free). In this code excerpt it is not difficult to identify the vulnerability, as execution is linear and the calls to free are within close proximity of each other, however in complex applications containing multiple threads and multiple source files there may be iterative methods which deal with memory management, containing complex conditional statements, and a slight error in logic or arithmetic can lead to an attempt at a double free.

A countermeasure to this form of memory corruption is to ensure that immediately after calling free() on a pointer, the pointers value is set to NULL. Any further calls to free will now no longer corrupt the heap, unless the pointer is modified in some way prior to the call.
<table>
<thead>
<tr>
<th>Name</th>
<th>Origin</th>
<th>Behaviour</th>
<th>Criteria Required</th>
<th>Impact</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strcpy() Overflow</td>
<td>Origins where a call to strcpy() is made where the size of the string in the source buffer is larger than that of the destination buffer.</td>
<td>A write operation exceeds the target memory allocated for it, and overwrites core variables on the stack such as ret or the base pointer.</td>
<td>Requires that the attacker have some level of control over the *src string. It is also possible to cause a bounds checked overflow to corrupt memory if the attacker is able to affect the allocation size of the buffer prior to the call.</td>
<td>Arbitrary code execution. Corruption of stack variables.</td>
<td>Bounds check all strcpy() functions dynamically based on sizeof(*dest)+1.</td>
</tr>
<tr>
<td>Strncpy() termination failure</td>
<td>Origins where the source buffer is of equal size or greater than the space allocated to store it, especially where n is also greater than the allocated number of bytes.</td>
<td>Variable based on context. Either creates string in memory that is not null terminated (and therefore corrupted in the eyes of functions like strcpy()) or overwrites EIP in a similar manner to strcpy() overflows.</td>
<td>Requires that either: - Attacker controls the size of buffer *dest - Attacker controls the integer N. -Attacker controls the size of the string in *src.</td>
<td>Arbitrary code execution and corruption of stack variables.</td>
<td>Throw an exception if strlen(*src) &gt; 1 &gt; sizeof(*dest). Determine the number of bytes to copy dynamically using sizeof.</td>
</tr>
<tr>
<td>Memcpy() Overflow</td>
<td>Origins where the number of bytes to be copied into *dest is greater than the size of *dest(.</td>
<td>A write operation exceeds the target memory allocated for it, and overwrites core variables on the stack such as ret or the base pointer.</td>
<td>Requires that either: - Attacker controls size of *src - Attacker controls content of *src and an integer &gt; sizeof(*dest) *dest is used as an argument to memcpy()</td>
<td>Arbitrary code execution and corruption of stack variables.</td>
<td>Ensure that the number of bytes to be copied is not greater than sizeof *dest prior to making the function call.</td>
</tr>
<tr>
<td>Free of non-dynamic memory</td>
<td>Origins where a buffer which has not been dynamically allocated is passed to free().</td>
<td>The free() accepts the pointer as a valid argument and attempts to modify the linked list structures which keep track of the heap with invalid values.</td>
<td>Requires that the attacker be able to manipulate the pointer in such a way to affect the free() procedure to cause later function calls to execute unexpected code.</td>
<td>Possible arbitrary code execution and corruption of the process heap.</td>
<td>Only free() pointers returned from malloc(), realloc() or calloc(). It is recommended to keep a “shadow table” of all pointers returned from these functions for verification.</td>
</tr>
<tr>
<td>Free after malloc() failure</td>
<td>Origins where a free() call is made to a pointer which was previously passed to malloc() where the memory allocation failed.</td>
<td>A free() attempts to remove the invalid pointer from the heap structures, and corrupts the process heap.</td>
<td>Requires that the attacker be able to manipulate the pointer in such a way to affect the free() procedure to cause later function calls to execute unexpected code.</td>
<td>Possible arbitrary code execution and corruption of the process heap.</td>
<td>Always check the return values of malloc(). If a call fails for any reason, set the value of the pointer to NULL, protecting the heap from corruption in subsequent calls to free().</td>
</tr>
<tr>
<td>Double Free()</td>
<td>Origins where a call to free() is made using a pointer which has already been free'd and never reallocated in between the calls to free.</td>
<td>A free() attempts to remove the invalid pointer from the heap structures, and corrupts the process heap.</td>
<td>Requires that the attacker be able to manipulate the pointer in such a way to affect the free() procedure to cause later function calls to execute unexpected code.</td>
<td>Possible arbitrary code execution and corruption of the process heap.</td>
<td>After freeing a pointer ensure that it is immediately set to NULL, as a free on a NULL pointer will have no effect on the heap, and prevent corruption.</td>
</tr>
<tr>
<td>Array out of bounds</td>
<td>Origins where a process attempts to read or write data outwith the bounds of a declared array.</td>
<td>The process reads data from other sections of memory. In an unbounded write, the write can overwrite data outside the array, causing memory corruption.</td>
<td>Requires that the attacker be able to control both the index of the array, and the contents of the write.</td>
<td>Arbitrary code execution and corruption of the stack/heap (depending on where the array was declared).</td>
<td>Check that the array index is not outwith the bounds of the array before performing any read or write operations.</td>
</tr>
<tr>
<td>Pointer reassignment to double free</td>
<td>Origins where a two malloc()ed pointers are reassigned to point to one of the pair. Later, when calling free(), the programmer fails to take the reassignment into account, and performs a double free on the same address.</td>
<td>A free() attempts to remove the invalid pointer from the heap structures, and corrupts the process heap.</td>
<td>Requires that the attacker be able to manipulate the pointer in such a way to affect the free() procedure to cause later function calls to execute unexpected code.</td>
<td>Possible arbitrary code execution and corruption of the process heap.</td>
<td>After freeing a pointer ensure that it is immediately set to NULL, as a free on a NULL pointer will have no effect on the heap, and prevent corruption.</td>
</tr>
<tr>
<td>Uncontrolled format string</td>
<td>Origins where a process makes a call to a function which requires a format specifier such as printf() without passing the format specifier.</td>
<td>The formatting function interprets any format specifiers in the string passed to it in the variable, and performs the appropriate methods.</td>
<td>Requires that the attacker be able to control the contents of the variable passed to the formatting function, and that no format specifier be provided by the developer.</td>
<td>Arbitrary code execution and disclosure of memory.</td>
<td>Always pass a format specifier to a function that requires it.</td>
</tr>
</tbody>
</table>
We tested a number of different techniques against known vulnerable code to identify which practices are best at identifying the memory corruption vulnerabilities covered by our taxonomy. Following this, a software testing methodology based on determined best practice is described, so that any security auditor may successfully audit an application with our taxonomy and yield positive results. This is an abstract representation of what is normally a very complex subject matter, due to the technical and polymorphic nature of software bugs.

To fully identify a range of software bugs, a direct analysis of the target software’s source code is the most efficient way of finding software vulnerabilities in generally robust code bases. A source code audit allows a security tester to develop a complete and complex understanding of the target software without much complication, a simple syntactical understanding of the programming language used by the developers being the only true requirement in order to be sufficient at identifying software vulnerabilities. The more developed the understanding of the target process, the more extensive the software audit will be. However, an issue arises when the target codebase increases beyond a certain size. At this point, the knowledge and skills required to sufficiently cover the entire codebase in adequate detail to identify possible security vulnerabilities becomes an incredibly complex task. The application written for the research was simple, yet commercial products with thousands of lines of code pose a complexity issue for the auditor. In order to combat this, source code analysis tools are used. However, even with a relatively simple target program like the one designed for the purposes of testing our taxonomy, it was determined that only a small proportion of the vulnerabilities are identified with these tools, and their effectiveness in comparison to a manual test is insufficient and unreliable for locating memory corruption bugs.

Typically, where source code is unavailable, a security consultant is still able to use alternative tools to effectively reverse engineer an understanding of the program equal to that achieved by a source code audit. However, the skill level required to successfully achieve a complex understanding of the program through decompilation is no small feat. The increased complexity results in an increase in the time investment required to identify a similar number and quality of vulnerabilities as more work is required to gain an equivalent knowledge of the behaviour of the target. In spite of their difficulty, a black box test is able to identify bugs which are impossible to identify at a source code level. Bugs introduced to a code base by a compiler mistake are able to be identified through reverse engineering as analysing a program at such a primitive level gives insight into the raw assembly instructions. This benefit only affects this type of test, as it is the only testing method which analyses the code of a binary after it has been through a compiler, and is object code ready for execution.

Both techniques can identify the same set of vulnerabilities, showing that given adequate time and skill to carry out, they can yield equivalent results. However, the ability to identify compiler level flaws in otherwise secure code, or the ability to successfully audit closed source applications to the same quality as an open source program are of invaluable benefit to software audits and so regardless of source availability it is beneficial to perform a black box audit in order to have the ability to identify the maximum number of software vulnerabilities in a minimal amount of time. An overview of our methodology which highlights the steps to vulnerability detection can be seen in Fig 2.

Fig 2. Overview of Methodology

Our proposed methodology combines a number of best practice techniques from static code analysis and reverse engineering.

1. The first step in the procedure requires a full source code audit. This should consist of:
   - A manual text based search through the codebase for the use of functions which may lead to memory corruption as identified in the taxonomy.
   - Identifying the variables used in the functions, and determine whether these variables could be used in an invalid way which may introduce memory corruption through reverse engineering as analysing a program at such a primitive level gives insight into the raw assembly instructions. This benefit only affects this type of test, as it is the only testing method which analyses the code of a binary after it has been through a compiler, and is object code ready for execution.

2. The second step uses reverse engineering techniques in order to identify compiler level vulnerabilities. Like the source code audit, the test should comprise of:
- A search through the codebase for use of functions which may lead to memory corruption
- Identify the variables used in the functions, and attempt to determine whether it is possible that these variables are used in an invalid way which could introduce memory corruption, by cross referencing the behaviour seen within the program against the taxonomy.

3. Incorporate countermeasures based on the mitigations described in the taxonomy for each identified vulnerability in the target.

IV. CONCLUSION

This research has identified a best practice approach to software testing for memory corruption vulnerabilities, based on analysis of the efficacy of existing techniques. Supplying software testers with a heuristical list of criteria to satisfy in order to identify vulnerabilities allows them to more accurately match the behavior identified in the software against the taxonomy of vulnerabilities. Importantly, and in contrast with many antipatterns, the taxonomy also provides the software tester with countermeasures as a solution to the vulnerabilities.

Only subjective evidence of the ease with which these particular security vulnerabilities could be recorded and disseminated using patterns currently exists. Future work will seek to study the efficacy of patterns as a means of documenting security concerns with a full-scale user study. The establishment of common (trial) pattern languages in the area would go a long way to enabling comparative studies.

REFERENCES


Using Attack Patterns to Develop a Penetration Testing Framework

Clive Blackwell
Computing and Communication Technologies
Faculty of Technology, Design and Environment
Oxford Brookes University,
Wheatley Campus, Wheatley, Oxford OX33 1HX
CBlackwell@brookes.ac.uk

Abstract – We show that both the automated tool and skill-based methods of penetration testing are unsatisfactory, because we need to provide comprehensive and complete evidence to clients about their weaknesses, and offer an adequate remediation plan to fix the critical ones. We use attack patterns to suggest a penetration-testing framework to help avoid the limitations of current approaches. There are further benefits in formalising attack patterns for penetration testing, which we propose to develop in future.

Keywords: attack pattern, penetration testing, design pattern, formal methods

1 INTRODUCTION TO ATTACK PATTERNS

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design, made famous in the software engineering community by the Design Patterns book [1]. It captures the context and structure of a general repeatable solution to a common problem archetype rather than its low-level implementation details. A design pattern represents desirable, intentional and expected system behaviour.

Attack patterns [2] are based on design patterns, and apparently emerged around 2001 in the security community, appeared first in the literature in Moore et al [3], before greater elaboration in Hoglund and McGraw’s book [4]. Subsequently, attack patterns were classified in the Common Attack Pattern Enumeration and Classification (CAPEC) schema [5] of which there are currently more than 400.

An attack pattern is analogous to a design pattern, as it describes how a particular type of attack is performed as a conceptual pattern, ignoring low-level implementation details. However, attack patterns are specified from the attacker’s viewpoint, and therefore represent undesirable, unintentional and unexpected behaviour. Here, the common problem targeted by the pattern represents the objective of the attacker, and the pattern’s solution represents common methods for performing the attack.

2 CRITIQUE OF PENETRATION TESTING

A common way to check the protection of an operational system is to attack it using a penetration test or vulnerability scan. Penetration tests actively exploit weaknesses, whereas vulnerability scans merely passively discover them, but we characterise them both as penetration testing for simplicity in this paper. Penetration testing can be characterised by two different methods at the ends of a spectrum of possibilities. We proceed to criticise the limitations of these extreme endpoints, at the risk of simplistic and crude stereotyping, in order to demonstrate clearly how attack patterns may help develop an effective penetration-testing framework.

At one end of the spectrum, penetration testing can be criticised as tool-driven, where a tool is executed to discover low-level system bugs and flaws automatically. The tool then provides an extensive report that gives the exploitability and significance of the discovered weaknesses. However, tools do not adequately emphasise and prioritise the crucial flaws that need remediation, because they do not understand the wider system context to determine which ones could cause serious damage, and others that are only minor annoyances. For example, access to sensitive data is hard to detect using an automated tool, because it does not understand its meaning, organisational purpose or possible damage if disclosed.

Our penetration-testing framework using attack patterns may offer a higher-level perspective to classify and characterise complete attacks, including their intent, surrounding context, and potential gains for the perpetrator and harm to the victim, rather than focussing on low-level bugs and weaknesses. An attack pattern can include the envisaged effects in the consequences section of its template and also possible unintentional side effects. For example, a data leak may also cause accidental loss of availability if the source data is removed or destroyed when it is copied.

The characterisation of an attack by an attack pattern may be classified into different subtypes in detailed child subpatterns inherited from the parent pattern, as in the CAPEC schema [5]. For example, the loss of different types of sensitive data may have diverse effects that can be captured in subpatterns of the parent data leakage pattern. The disclosure of passwords allows system access, loss of account information may allow fraud, the misuse of personal data allows impersonation, and compromise of intellectual property can be used to create competitive products and services, which can all be captured in child subpatterns for each data type. At the very least, the better understanding of sensitive data meaning, purposes and weaknesses may guide the penetration tester to determine the significance of breaches discovered with automated tools, and therefore help to prioritise remediation measures.

At the other end of the scale, skilled penetration testers have a theoretical and empirical rationale and model for their activities based on their prior knowledge and experience, and check for significant weaknesses
based on the system context, goals, critical functions and likely weaknesses. A skilled penetration tester may be able to find many issues that an automated tool would not detect, by effectively formulating and testing higher-level pattern-matching hypotheses.

However, their mental models may lack a valid foundation, coherent organisation and completeness. For example, they may try a particular attack in one or two ways that have worked in the past, but not try all variations for which an automated tool would be superior. In addition, they may discover an attack, such as the lack of adequate protection for sensitive data, but they would be aided by an attack pattern that includes detailed information such as the purpose of the data, its meaning and issues resulting from unauthorised disclosure.

We conclude that both the automated tool and skill-based methods are unsatisfactory, because the penetration tester needs to provide convincing, comprehensible and complete evidence to clients about their weaknesses, and offer an adequate, timely and cost-effective remediation plan to fix the critical ones.

3 PROPOSED PENETRATION TESTING FRAMEWORK

We propose a penetration-testing framework based on attack patterns to formulate a comprehensive test plan that exhaustively covers all attack occurrences. An attack pattern provides a space of possible attacks that scope or bound individual attack instances. We can instantiate attack patterns by binding parameters to possible values, and categorising their behaviour by grouping attack vectors that display the same features into equivalence classes.

This could lead to a testing framework that can provide comprehensive coverage, and address some of the limitations in both the skill-based and automated approaches to penetration testing described above. The proposed framework is similar to automated tool-driven testing, but at a more meaningful scale of analysing complete attacks rather than atomic exploits. The resulting penetration-testing plan would offer a comprehensible view of the tests, thereby also helping to make the results actionable by the client. The use of attack patterns for black-box or white-box testing of software [6] can also be used for penetration testing of systems, which we intend to develop in further work.

The framework may help to hypothesise novel attacks by instantiating attack patterns by allocating possible, but never previously observed values to parameters, or bringing unseen combinations of variables together in a novel attack instance. Another possibility is to use existing attack instances as exemplars, and then adapt or extend them by changing their parameters, a bit like fuzzing [7], except in a meaningful and relevant way to the attack being executed.

Composition of attack patterns is also relevant and can bootstrap off existing work in design patterns [8]. Attack patterns could be composed to create complex, structured and realistic tests that are similar to real attacks that need to compromise several weaknesses to achieve their goals.

We might hope that our framework could lead to a systematic theory of penetration testing and corresponding remediation. We may be able to show that certain attacks cannot occur, by testing all the variations of the attack pattern.

4 FORMALISATION

Eventually, we might hope this could lead to a formal theory of penetration testing and corresponding defensive controls, by adapting and extending the formalisation of design patterns [9] to attack patterns. For example, if both the attack and defensive systems have a formal representation, we may be able to prove that certain attacks are impossible, assuming the system realises the formal model.

Admittedly, formal models do not represent system security adequately, as any proof is of a model and not of the system, so therefore we propose a ‘belt and braces’ process. We can indicate the assumptions of such proofs using the prerequisites section of attack patterns. A formal model of attack patterns may highlight the most relevant penetration tests to discover if the system is operating according to its underlying assumptions. We can formulate a comprehensive penetration-testing plan that classifies and covers all possible values and actions of the attack systematically to test system assumptions that may not be realised in practice, and thus check the prerequisites of the corresponding proof.

5 CONCLUSIONS AND FURTHER WORK

We showed that both the automated tool and skill-based methods of penetration testing are unsatisfactory, because we need to provide comprehensible and complete evidence to clients about their weaknesses, allowing them to take effective remediation measures. We suggested using attack patterns to help develop a penetration-testing framework to address the current issues of validity, completeness and comprehensibility in penetration testing.

We can also apply the idea of regression testing [10] to penetration testing using attack patterns. We could develop a protocol for repairing system weaknesses that attempts to ensure that no new flaws are introduced by repeating the penetration test with relevant new values determined from the attack patterns.

Eventually, we might hope to develop a systematic theory of penetration testing, where we may be able to show that certain attacks cannot occur by testing all the variants of the attack pattern. Formalisation will help especially if it is combined with the empirical testing of underlying assumptions in the penetration test.

REFERENCES

Composition Patterns of Hacking

Sergey Bratus*, Julian Bangert*, Alexandar Gabrovsky*, Anna Shubina*, Daniel Bilar†, and Michael E. Locasto‡

†Dartmouth College
‡Siege Technologies
§The University of Calgary

Abstract—You do not understand how your program really works until it has been exploited. We believe that computer scientists and software engineers should regard the activity of modern exploitation as an applied discipline that studies both the actual computational properties and the practical computational limits of a target platform or system.

Exploit developers study the computational properties of software that are not studied elsewhere, and they apply unique engineering techniques to the challenging engineering problem of dynamically patching and controlling a running system. These techniques leverage software and hardware composition mechanisms in unexpected ways to achieve such control. Although unexpected, such composition is not arbitrary, and it forms the basis of a coherent engineering workflow. This paper contains a top-level overview of these approaches and their historical development.

I. INTRODUCTION

When academic researchers study hacking, they mostly concentrate on two classes of attack-related artifacts: “malicious code” (malware, worms, shellcode) and, lately, “malicious computation” (exploits via crafted data payloads containing no native code, a popular exploit technique). We argue that studying just these two classes is an insufficient means of making progress toward the goal of more fundamentally secure software because this perspective does not include a notion of composing the attacker computation with the native computer computation of the target. We claim that such composition is the source of the most powerful and productive concepts and methodologies that emerge from exploitation practice.

When researchers focus on attacker artifacts alone, they frequently miss an important point of successful exploitation: the exploited system needs to remain available and reliably usable for the attacker.

In order to support this assertion and further discuss exploitation, we need to make an important terminological point. The word hacking is used to refer to all kinds of attacks on computer systems, including those that merely shut down systems or otherwise prevent access to them (essentially achieving nothing that could not be achieved by cutting a computer cable). Many activities labeled as hacking lack sophistication. In this paper we discuss not hacking, but exploitation or exploit programming. We take exploitation and exploit programming to mean subverting the system to make it work for the attacker – that is, lend itself to being programmed by the attacker. Exploiters are less interested in BSODs, kernel panics, and plain network DOS attacks that merely result in a DoS on the target and cannot otherwise be leveraged and refined to take control over the system rather than disabling it.

Not surprisingly, preventing a disabling crash and subsequently “patching up” the target into a stable running state requires significantly more expertise and effort than, say, a memory-corrupting DoS. By achieving this state exploiters demonstrate a sophisticated understanding of the target platform.

In this paper we review a series of classic exploitation techniques from the perspective of composition (specifically, composition as the basic unit of activity of an engineering workflow, whether that workflow is a traditional software engineering workflow or a workflow focused on engineering an exploit). Many of these have been extensively described and reviewed from other perspectives; however, their compositional aspect is still treated as ad hoc, and has not, as far as we know, been the subject of systematic analysis.

We posit that such analysis is required for designing defensible systems (see Section IV). The practical properties of composition in actual computer systems uncovered and distilled by hacker research have often surprised both designers and defenders. We believe that the relevant methods here must be cataloged and generalized to help approach the goal of secure composition in future designs.

II. A TALE OF TWO ENGINEERING WORKFLOWS

Hacking, vulnerability analysis, and exploit programming are generally perceived to be difficult and arcane activities. The development of exploits is seen as something unrepeatable and enabled only by some unfortunate and unlikely combination of events or conditions. Almost by definition, something as imbued with arbitrary chance cannot or should not be an engineering discipline or workflow. Popular perception casts these activities as requiring specialized cross-layer knowledge of systems and a talent for “crafting” input.

This paper asserts that what seems arcane is really only unfamiliar. In fact, although it may be difficult to conceive of

1Discussed in the hacker community since at least early 2000s, see [2] for a brief history sketch. These artifacts were brought to the attention of academia by Shacham [30], [24].

2It also serves as an excellent teaching aid in advanced OS courses; see, e.g., [25].

3Since our focus is on composition, we do not distinguish between techniques used by rootkits vs. exploits. Indeed, rootkits, even when introduced at higher privilege, face similar context limitation obstacles as exploits.
exploit development as anything other than fortunate mysticism, we argue that its structure is exactly that of a software engineering workflow. The difference emerges in the specific constructs at each stage, but the overall activities remain the same. A software developer engineers in terms of sequences of function calls operating on abstract data types, whereas an exploit developer engineers in terms of sequences of machine-level memory reads and writes. The first one programs the system in terms of what its compile-time API promises; the other programs it in terms of what its runtime environment actually contains.

This section contains a brief comparison of these two engineering workflows. We do so to help give a conceptual frame of reference to the enumeration of exploit techniques and composition patterns detailed in Section III.

The main difference between the two workflows is that the exploit engineer must first recover or understand the semantics of the runtime environment. In either case, programming is composition of functionality.

In the “normal” workflow of software engineering, the programmer composes familiar, widely-used libraries, primitive language statements (repetition and decision control structures), and function calls to kick input data along a processing path and eventually produce the result dictated by a set of functional requirements.

In the exploit workflow, the reverser or exploit engineer attempts to build this programming toolkit from scratch: the languages and libraries that the software engineer takes for granted are not of direct use to the exploit developer. Instead, these elements define a landscape from which the exploit developer must compose and create his own toolkit, language primitives, and component groups. The first job of the vulnerability analyst or reverse engineer is therefore to understand the latent functionality existing in runtime environments that the software engineer either neglects or does not understand.

A. The Software Engineer

Based on functional requirements, a software engineer’s goal is to cause some expected functionality happen. In essence, this kind of programming is the task of choosing a sequence of library calls and composing them with language primitives like decision control structure and looping control structures. Data structures are created to capture the relevant properties of the system’s input; this structure usually dictates how processing (i.e., control flow) occurs.

A software engineer follows roughly this workflow path:

1) design and specify data types
2) design data flow relationships (i.e., an API)
3) write down source code implementing the data types and API
4) ask compiler and assembler to translate code
5) ask OS to load binary, invoke the dynamic linker, and create memory regions
6) run program according to the control flow as conceived in the source level

In this workflow, we can see the software engineer engaged in: memory layout, specifying control flow, program construction, program delivery (loading) and translation, and program execution. As we will see below, the exploit engineer engages in much the same set of tasks.

The software engineer’s goal is to bring order to a composition of procedures via compilation and assembly of machine code. One does this through toolchains, design patterns, IDEs, and popular languages — the software engineer therefore does not need to relearn the (public) semantics of these operations every time he prepares to program.

These conventions are purely an effort–saving device aimed at increasing productivity by increasing the lines of code and features implemented in them. These patterns, tools, and aids reduce the level of thought required to emit a sequence of function calls that satisfy the functional requirements. They are an effort to deal with complexity. The goal of software engineers in dealing with complexity is to eliminate or hide it.

B. The Exploit Engineer

In contrast, exploit engineers also deal with complexity, but their goal is to manipulate it — expressiveness, side effects, and implicit functionality are a collective boon, not a bane. Any operations an exploit engineer can get “for free” increase his exploit (i.e., weird machine programming) toolkit, language, or architecture. A software engineer attempts to hide or ignore side effects and implicit state changes, but the very things encouraged by traditional engineering techniques like “information hiding” and encapsulation on the other side of an API become recoverable primitives for a reverser or exploit engineer.

The main difference in the workflows is the preliminary step: you have to learn on a case by case or scenario by scenario basis what “language” or computational model you should be speaking in order to actually begin programming toward a specific functional end. Based on some initial access, the first goal is to understand the system enough to recover structure of “programming” primitives. The workflow is thus:

1) identify system input points
2) recapture or expose trust relationships between components (functions, control flow points, modules, subroutines, etc.)
3) recover the sequencing composition of data transformations (enumerate layer crossings)
4) enumerate instruction sequences / primitives / gadgets
5) program the process address space (prepare the memory image and structure)
6) deliver the exploit

In this workflow, we can see the exploit engineer engaged in: recovering memory layout, specifying control flow, program construction, program delivery (loading) and translation, and program execution. Unlike the software engineering workflow, the delivery of an exploit (i.e., loading a program) can be mixed up and interposed with translation of the program and preparation of the target memory space. Even though
these activities might be more tightly coupled for an exploit developer, much of the same discipline remains.

One major challenge exists for the exploit engineer: recovering the unknown unknowns. Although they can observe side effects of mainline execution or even slightly fuzzed execution, can they discover the side effects of “nearly” dormant or latent “normal” functionality (e.g., an internationalization module that is never invoked during normal operation, or configuration code that has only been invoked in the “ancient past” of this running system)? This challenge is in some sense like the challenge a software engineer faces when exploring a very large language library (e.g., the Java class library API).

III. PATTERNS

a) Exploitation as programming “weird machines”: Bratus et al. [2] summarized a long-standing hacker intuition of exploits as programs, expressed as crafted inputs, for execution environments implicitly present in the target as a result of bugs or unforeseen combination of features (“weird machines”), which are reliably driven by the crafted inputs to perform unexpected computations. More formally, the crafted inputs that constitute the exploit drive an input-accepting automaton already implicitly present in the target’s input-handling implementation, its sets of states and transitions owing to the target’s bugs, features or combinations thereof. The implicit automaton is immersed into or is part of the target’s execution environment; its processing of crafted input is part of the “malicious computation” – typically, the part that creates the initial compromise, after which the exploiter can program the target with more conventional means. The crafted input is both a program for that automaton and a constructive proof of its existence. Further discussion from the practical exploit programming standpoint can be found in Dullien [7], from a theory standpoint in Sassaman [28].

In the following items, we focus on one critical aspect of the implicit exploit execution environments and the computations effected in them by exploit-programs: they must reliably coexist with the native, intended computations both for their duration and in their effects, while their composition is done in contexts more limited and lacking critical information as compared to the system’s intended scenarios. This is far from trivial on systems where state that is “borrowed” by the exploit computation thread of control is simultaneously used by others, and involves dissecting and “slimming down” interfaces to their actual implementation primitives and finding out unintended yet stable properties of these primitives.

b) Recovering Context, Symbols, and Structure: To compose its computation with a target, an exploit must refer to the objects it requires in its virtual address space (or in other namespaces). In essence, except in the most trivial cases, a “name service” of a kind (ranging from ad-hoc to the system’s own) is involved to reconstruct the missing information.

Early exploits and rootkit install scripts relied on hard-coded fixed addresses of objects they targeted, since back then memory virtual space layouts were identical for large classes of targets4. As targets’ diversity increased, naturally or artificially (e.g., OpenWall, PaX, other ASLR), exploits progressed to elaborate address space layout reconstruction schemes and co-opting the system’s own dynamic linking and/or trapping debugging.

Cesare [5] describes the basic mechanism behind ELF linking – based on little more that careful reading of the ELF standard. However, it broke the opacity and resulted in an effective exploit technique, developed by others, e.g., [29]. In [16] mayhem builds on the same idea by looking into the significance and priority of ELF’s .dynamic symbols. Nergal [17] co-opted Linux’s own dynamic linker into anROP5 crafted stack frame-chaining scheme, to have necessary symbols resolved and libraries loaded. Oakley [18] showed how to co-opt the DWARF-based exception handling mechanism.

Skape [31] takes the understanding of ELF in a different direction by showing how its relocation mechanism works and how that could be used for unpacking a binary.

c) Preparing Vulnerable System State: Earlier classes of exploits leveraged conditions and configurations (such as memory allocation of relevant objects) present in the target’s state through all or most runs. Subsequent advancements such as Sotiroy’s [32] demonstrated that otherwise non-exploitable targets can have their state carefully prepared by way of a calculated sequence of requests and inputs for an exploitable configuration to be instantiated.

This pattern of pre-compositional state-construction of targets is becoming essential, as protective entropy-injecting techniques prevent setting up an effective “name service”. Recent examples [22], [11] show its applications to modern heaps (the former for the Windows low fragmentation heap), in presence of ASLR and DEP. Moreover, this method can target the injected entropy directly, by bleeding it from the target’s state (e.g., [8]).

d) Seeing through Abstraction: Developers make use of abstractions to decrease implementation effort and increase code maintainability. However, abstractions hide the details of their implementation and as they become part of a programmers daily vocabulary, the implementation details are mostly forgotten. For example, few programmers worry about how a function call is implemented at the machine level or how the linking and loading mechanisms assign addresses to imported symbols.

Exploit engineers, however, distill abstractions into their implementation primitives and synthesize new composition patterns from them. Good examples of this are found in [17], who modifies the return addresses on the stack to compose existing code elements into an exploit, and the LOCATE [31] packer which obfuscates binary code by using the primitives

4This fact was not well understood by most engineers or academics, who regarded below-compiler OS levels as unpredictable; Stephanie Forrest deserves credit for putting this and other misconceptions into broader scientific perspective.

5Which it pre-dates, together with other hacker descriptions of the technique, by 5-7 years.
for dynamic linking.

e) Scoping Out The Devil’s Bargain:
Wherever there is modularity there is the potential
for misunderstanding; hiding information implies a
need to check communication. A. Perlis

When a software architect considers how much context to
pass through an interface, he has to bargain with the devil.
Either a lot of context is passed, reducing the flexibility of
the code, or too little context is preserved and the remaining
data can no longer be efficiently validated by code operat-
ing on it, so more assumptions about the input have to be
trusted. Exploiters explore this gap in assumptions, and distill
the unintended side-effects to obtain \textit{primitives}, from which
weird machines are constructed [23], [9], [7]. We posit that
understanding this gap is the way to more secure API design.

f) Bit path tracing of cross-layer flows:
When an explo-

er studies a system, he starts with bit-level description of
its contents and communications. Academic textbooks and
user handbooks, however, typically do not descend to bit level
and provide only a high-level description of how the system
works. A crucial part of such bit-level description is the flow
of bits between the conceptual design layers of the system: i.e.
a binary representation of the data and control flow between
layers.

Constructing these descriptions may be called the corner-
stone of the hacker methodology. It precedes the search for
actual vulnerabilities and may be thought of as the modeling
step for constructing the exploit computation. The model may
ignore large parts of the target platform but is likely to punctil-
iously describe the minutiae of composition mechanisms that
actually tie the implementations of the layers together.

For example, the AlephOne Phrack article [19] famous for
its description of stack buffer overflows also contained
a bit-level description of UNIX system calls, which for many
readers was in fact their first introduction to syscall mech-
anisms. Similarly, other shellcode tutorials detailed the data
flow mechanisms of the target’s ABIs (such as various calling
conventions and the structure of libraries). In networking,
p particular attention was given to wrapping and unwrapping
of packet payloads at each level of the OSI stack model, and
libraries such as libnet and libdnet were provided for emulating
the respective functionality throughout the stack layers.

What unites the above examples is that in all of them
exploiters start analyzing the system by tracing the flow of
bits within the target and enumerating the code units that
implement or interact with that flow. The immediate benefits
of this analysis are at least two-fold: locating of less known
private or hidden APIs and collecting potential exploitation
primitives or “cogs” of “weird machines”, i.e. code fragments
on which crafted data bits act in predictable way.

Regardless of its immediate benefits, though, bit-level cross-
layer flow descriptions also provide useful structural descrip-
tions of the system’s architecture, or, more precisely, of the
mechanisms that underlie the structure, such as the library
and loadable kernel functionality, DDKs, and network stack
composition.

For instance, the following sequence of Phrack articles
on Linux rootkits is a great example of deep yet concise
coverage of the layers in the Linux kernel architecture: \textit{Sub
proc_root Quando Sumus (Advances in Kernel Hacking)} [21]
(VFS structures and their linking and hijacking), \textit{5 Short
Stories about execve (Advances in Kernel Hacking II)} [20]
(driver/DDK interfaces, different binary format support), and
\textit{Execution path analysis: finding kernel based rootkits} [27]
(instrumentation for path tracing). Notably, these articles at
the cusp where three major UNIX innovations meet: VFS, kernel
state reporting through pseudo-filesystems (e.g., \texttt{/proc}), and
support for different execution domains/ABI. These articles
described the control and data flows through a UNIX kernel’s
component layers and their interfaces in great detail well
before tools like DTrace and KProbes/SystemTap brought
tracing of such flows within common reach.

It is worth noting that the ELF structure of the kernel binary
image, the corresponding structure of the kernel runtime,
and their uses for reliably injecting code into a running
kernel (via writing \texttt{/dev/kmem} or via some kernel memory
corruption primitive). In 1998, the influential \textit{Runtime kernel
kmem patching} [4] made the point that even though a kernel
may be compiled without loadable kernel module support, it
still is a structured runtime derived from an ELF image file,
in which symbols can be easily recovered, and the linking
functionality can be provided without difficulty by a minimal
userland “linker” as long as it has access to kernel memory.
Subsequently, mature kernel function hooking frameworks
were developed (e.g., \textit{IA32 Advanced function hooking} [14]).

Dynamic linking and loading of libraries (shared binary
objects) provide another example. This is a prime example
of composition, implicitly relied upon by every modern OS
programmer and user, with several supporting engineering
mechanisms and abstractions (ABI, dynamic symbols, call-
ing conventions). Yet, few resources exist that describe this
key mechanism of interposing computation; in fact, for a
long time hacker publications have been the best resource
for understanding the underlying binary data structures (e.g.,
\textit{Backdooring binary objects} [13]), the control flow of dynamic
links (e.g., \textit{Cheating the ELF} [34] and \textit{Understanding Linux
ELF RTLD internals} [15]), and the use of these structures for
either binary infection (e.g., the original \textit{Unix ELF parasites and
virus}) or protection (e.g., \textit{Armouring the ELF: Binary
encryption on the UNIX platform} [10]).

A similar corpus of articles describing the bit paths and
layer interfaces exists for the network stacks. For the Linux
kernel stack, the \textit{Netfilter} architecture represents a culmi-
nation of this analysis. By exposing and focusing on specific hooks
(tables, chains), Netfilter presents a clean and concise model
of a packet’s path through the kernel; due to this clarity it
became both the basis of the Linux’s firewall and a long series of
security tools.

Not surprisingly, exploitative modifications of network
stacks follow the same pattern as other systems rootkits.
\textit{Passive Covert Channels Implementation in Linux Kernel} [26]
is a perfect example: it starts with describing the interfaces
traversed on a packet’s path through the kernel (following the Netfilter architecture), and then points out the places where a custom protocol handler can be inserted into that control flow, using the stack’s native protocol handler interfaces.

4) “Trap-based programming and composition”. In application programming, traps and exceptions are typically not treated as “first-class” programming primitives. Despite using powerful exception-handling subsystems (such as GCC’s DWARF-based one, which employs Turing-complete bytecode), applications are not expected to perform much of their computation in traps or exceptions and secondary to the main program flow. Although traps are obviously crucial to systems programming, even there the system is expected to exit their handlers quickly, performing as little and as simple computation as possible, for both performance and context management reasons.

In exploit programming and reverse engineering (RE), traps are the first-class programming primitives, and trap handler overloading is a frequently used technique. The target platform’s trap interfaces, data structures, and contexts are carefully studied, described, and modeled, then used for reliably composing an exploit or a comprehension computation (i.e., a specialized tracer of debugger) with the target.

The tracing and debugging subsystems in OS kernels have long been the focus of hacker attention (e.g., Runtime Process Infection [1] for an in-depth intro to the ptrace() subsystem). Not surprisingly, hackers are the leading purveyors of specializations debuggers, such as dumbBug, Rasta Debugger, and the Immunity debugger to name a few.

For Linux, a good example is Handling Interrupt Descriptor Table for fun and profit [12], which serves as both a concise introduction to the x86 interrupt system and its use on several composition-critical kernel paths, as well as its role in implementing various OS and debugging abstractions (including system calls and their place in the IDT). This approach was followed by a systematic study of particular interrupt handlers, such as the Hijacking Linux Page Fault Handler [3].

Overloading the page fault handler in particular has become a popular mechanism for enforcing policy in kernel hardening patches (e.g., PaX and OpenWall); however, other handlers have been overloaded as well, providing, e.g., support for enhanced debugging not relying on the kernel’s standard facilities – and thus not conflicting with them and not registering with them, to counteract anti-debugging tricks. Since both rootkits (e.g., the proof-of-concept DR Rootkit that uses the x86 debug registers exclusively as its control flow mechanism) and anti-RE armored applications (e.g., Skype, cf. Vanilla Skype [6]; also, some commercial DRM products). In particular, the Rasta Debugger demonstrates such “unorthodox debugging” trap overloading-based techniques.

Notably, similar trap overloading techniques are used to expand the semantics of classic debugger breakpoint-able events. For instance, OllyBone6 manipulated page translation to catch an instruction fetch from a page just written to, a typical behavior of a malware unpacker handing execution to the unpacked code. Note the temporal semantics of this composed trap, which was at the time beyond the capabilities of any debugger. A similar use of the x86 facilities, and in particular the split instruction and data TLBs was used by the Shadow Walker [33] rootkit to cause code segments loaded by an antivirus analyzer to be fetched from a different physical page than the actual code, so that the analyzer could receive innocent data – a clever demonstration of the actual vs assumed nature of x86 memory translation mechanism.

IV. Conclusion

Exploit engineers will show you the unintended limits of your system’s functionality. If software engineers want to reduce this kind of latent functionality, they will have to begin understanding it as an artifact that supports the exploit engineer’s workflow.

Software engineers should view their input data as “acting on code”, not the other way around; indeed, in exploits inputs serves as a de-facto bytecode for execution environments that can be composed from the elements of their assumed runtime environment. Writing an exploit — creating such bytecode — is as structured a discipline as engineering “normal” software systems. As a process, it is no more arcane or unapproachable than the ways we currently use to write large software systems.

Yet, a significant challenge remains. If, as hinted above, we want to have a practical impact on the challenge of secure composition, can we actually train software engineers to see their input parameters and data formats as bytecode even as they specify it? Even as they bring it into existence, where it is by definition partially formulated, can they anticipate how it might be misused? Is this constant and frequent self-check worth the effort, or should software engineers first build a system without regard to analyzing anti-security composition patterns?

Acknowledgment

This paper would not be possible without the efforts of Felix Lindner, Hugo Fortier, Bruce Potter, David Hulton, Enno Rey and other organizers of conferences where many of the results we discussed have been presented, and many discussions took place.

References

Author Index

A
Amorim, Renato, 47
Appleby, Susan, 21
Asim, Muhammad, 17, 39

B
Bangert, Julian, 80
Bayley, Ian, 6
Bilar, Daniel, 80
Blackwell, Clive, 35, 77
Bratus, Sergey, 80

C
Carter, Colin, 21
Chris, P., 21
Coull, Natalie, 72

E
Ekblom, Paul, 26

F
Faily, Shamal, 10
Ferguson, Ian, 72

G
Gabrovsky, Alexandar, 80
Glisson, William Bradley, 30
Griffiths, Nathan, 52

H
Hutchison, David, 12, 67

K
Kearney, Paul, 62
Komisarczuk, Peter, 47

L
Lallie, Harjinder, 52
Lamb, David, 39
Llewellyn-Jones, David, 17, 39, 67
Locasto, Michael, 80
Lyle, John, 10

M
Mauthe, Andreas, 12
Merabti, Madjid, 17, 39, 67
Milne, Allan, 72
Mitchell, Faye, 43

P
Parkin, Simon, 10

R
Renaud, Karen, 30
Ruskov, Martin, 26

S
Sasse, M. Angela, 26
Schaeffer-Filho, Alberto E., 12, 67
Shi, Qi, 17, 39, 67
Shirazi, Noor-Ul-Hassan, 67
Shubrina, Anna, 80
Sinclair, David, 62
Smith, Paul, 12
Storer, Tim, 30

W
Wagner, Sebastian, 62
Walker, Nick, 72

Z
Zhou, Bo, 17, 39
Zhu, Hong, 1