Retaining Consistency for Knowledge-based Security Testing

Andreas Bernauer*[1], Josip Bozic[2], Dimitris E. Simos**[3], Severin Winkler[1], and Franz Wotawa[2]

[1] Security Research, Favoritenstrasse, Vienna, 1040, Austria
[2] Techn. Univ. Graz, Inffeldgasse 16b/2, 8010 Graz, Austria
[3] SBA Research, Favoritenstrasse, Vienna, 1040, Austria
{abernauer,swinkler}@securityresearch.at
{jbozic,wotawa}@ist.tugraz.at
dsimos@sba-research.org

Abstract. Testing of software and systems requires a set of inputs to the system under test as well as test oracles for checking the correctness of the obtained output. In this paper we focus on test oracles within the domain of security testing, which require consistent knowledge of security policies. Unfortunately, consistency of knowledge cannot always be ensured. Therefore, we strongly require a process of retaining consistencies in order to provide a test oracle. In this paper we focus on an automated approach for consistency handling that is based on the basic concepts and ideas of model-based diagnosis. Using a brief example, we discuss the underlying method and its application in the domain of security testing. The proposed algorithm guarantees to find one root cause of an inconsistency and is based on theorem proving.

Keywords: model-based diagnosis; root cause analysis; testing oracle

1 Introduction

Ensuring safety and security of today's software and systems is one of the big challenges of industry but also of society especially when considering infrastructure and the still increasing demand of connecting services and systems via the internet and other means of communication. Whereas safety concerns have been considered as important for a longer time, this is not always the case for security concerns. Since security threads often are caused because of bugs in software or design flaws, quality assurance is highly required.

Testing software and systems is by far the most important activity to ensure high quality in terms of a reduced number of post-release faults occurring after

* Authors are listed in alphabetical order.

** This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. This Programme is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission.

the software deployment. Hence, there is a strong need for testing security related issues on a more regular bases. Unfortunately, testing with the aim of finding weaknesses that can be exploited for an attack is usually a manual labor and thus expensive. Therefore, automation of software testing in the domain of software security is highly demanded.

In order to automate testing we need input data for running the *system under test* (SUT) and a test oracle that tells us whether the observed system's output is correct or not. There are many techniques including combinatorial testing [2] that allows for automatically deriving test input data for SUTs based on information about the system or its inputs. Unfortunately, automating the test oracle is not that easy, which is one reason for manually carrying out testing. In order to automate the test oracle, knowledge about the SUT has to be available that can be used for judging the correctness of the received output. In the domain of security the security policies have to be available in a formal form. To solve this problem, we suggest a framework for formalizing the knowledge base, which comprises the two different parts: (1) general security knowledge, and (2) SUT specific knowledge. We distinguish these parts in order to increase reuse. The general security knowledge might be used for testing different systems while the SUT specific knowledge only comprise additional clarifications and requirements to be fulfilled by one SUT. For example, in web-based applications a login should expire at least within a certain amount of time, say 60 minutes. In a particular application the 60 minutes might be too long and a specific policy overrides this value to 30 minutes.

Unfortunately, distinguishing this two kinds of knowledge might lead to trouble. Let us discuss an example. In general no user should be allowed for adding another user to the system with the exception of a user with superuser privileges. This piece of knowledge is more or less generally valid. A specific SUT might have here a restriction, i.e., stating that some users with restricted administrator rights are allowed to add new users. When assuming that such users do not have all rights of a superuser we are able to derive a contradiction. Another example would be that a text field should never be allowed for storing Javascript in order to avoid cross-site scripting attacks (see [7] for an introduction into the area of security attacks and how to avoid them). In web application for developing web pages such a restriction does not make any sense and has to be relaxed at least for certain text fields. All these examples indicate that there is a strong need to resolve inconsistencies between general security policies and custom policies used for specific applications. In order to increase automation retaining consistency has to be performed as well in an automated way.

The challenge of handling inconsistencies in knowledge bases has been an active research question in artificial intelligence for decades. Model-based diagnosis [4, 9, 6] offers means for removing inconsistencies via making assumptions explicit. In this paper, we adopt the underlying ideas in order to handle inconsistencies in knowledge bases used for implementing as test oracles. Note that the presented approach is tailored to handle security testing where general and custom polices together are used for implementing a test oracle. Our approach

is not restricted to a particular test suite generation method providing that the used testing method delivers test input for the SUT and that there is an oracle necessary in order to distinguish faulty from correct SUT's output.

This paper is organized as follows. In the next section we discuss the underlying problem in more detail using a running example. Afterwards, we introduce the consistency problem and provide an algorithm for solving this problem. Finally, we discuss related research, conclude the paper, and give some directions for future research.

2 Running Example

In order to illustrate the problem we want to tackle in this paper we discuss a small example in more detail. Let us consider a web application having an user authentication page where each user has to login in order to access the system. In this case the initial state is the login screen. After entering the correct user information, i.e., user name and password, the system goes to the next state, e.g., a user specific web page. The action here would be the login. The facts that hold in the next state are the user name and the information that the user logged in successfully.

Testing such a SUT requires the existence of test cases. We assume that there is a method of generating such test cases. This can be either done manually or automatically using testing methods. We call the resulting set of test cases a *test suite* for the SUT. When applying a certain test case t to the SUT there is now the question of determining whether the run is correct or not. In case a SUT behaves as expected when applying a test case t we call t a *passing test case*. Otherwise, t is a *failing test case*. But how to classify test cases as failing or passing? To distinguish test cases we introduce an application's *policy* P, which is a consistent set of first order logic rules. When using such a policy P for classifying a test case t we ask a theorem prover whether the output of the SUT after applying t is consistent with the policy P. If this is the case, the test case t is a passing test case, and a failing one, otherwise. Hence, consistency checking based on P can be considered as our test oracle.

An example for a policy P for our small example application would be: For every user access to the user specific web page is only possible if and only if the authentication is successful. The authentication is successful if and only if the user exists in the user database and the given password is equivalent to the one stored in the database. Some other examples of the rules that form a policy are the following:

1. A user (except the superuser) must not add other users to the application.
2. The database is accessed by the application only but never directly by a user.
3. A user must not be able to overwrite any configuration file.

Note that policies describe the available knowledge of the domain and a particular application. As already said we distinguish two kinds of policy knowledge:

the general domain specific knowledge and the SUT specific knowledge, i.e., a policy P comprises two parts: the general policy P_G and the custom policy P_C, i.e., $P = P_G \cup P_C$.

When separating P into two parts immediately two problems arise. First, the general policy might be in contradiction with the custom policy. In the introduction we already discussed an example. Second, there might be parts of the general policy that are not applicable for a certain SUT. For example, consider an application that uses a database to store data. A policy for this case must contain all rules that defines the database's storage functionality. In the same example, if the application does not use a database to store data, then the policy does not need to contain rules for this functionality. The latter problem is not severe because a policy that is not used in an application would never lead to a contradiction with any facts of any state of the application. Thus such a policy would never cause a test case to fail.

Let us discuss the consistency problem in more detail. For this purpose we formalize the example given in the introduction stating that ordinary users are not allowed to add new users. Only superusers might add new users. In order to formalize this general policy P_G we introduce the predicates `user(X)`, `superuser`, and `add_user`. The first order logic rules using the syntax of Prover9 [8] with the Prolog-style option for variables and constants would be:

```
all X (superuser(X) -> add_user(X)). all X ((user(X) &
-superuser(X)) -> -add_user(X)).
```

Moreover, we would also add a rule stating that superusers are also users. This is necessary in order to ensure that superusers have at least all privileges of users and sometimes additional more.

```
all X (superuser(X) -> user(X)).
```

Obviously, these three rules are consistent. We can check this by running a theorem prover like Prover9 using the rules. If there is no contradiction the theorem prover would not find any proof.

Let us now formalize a custom policy P_C. There we introduce a new kind of user that is able to add new users. We call this user, a user with administration rights and introduce a new predicate `admin` to represent such a user formally.

```
all X (admin(X) -> user(X)). all X (admin(X) -> add_user(X)).
```

In addition we add a specific user with administration rights *Julia* to the application and an ordinary user *Adam*. Hence, we add the following facts to the custom policy P_C:

```
admin(julia). -superuser(julia). user(adam).
```

Note that we explicitly have to state that *Julia* is a user with administration rights only (and not a superuser). This might be avoided when adding more knowledge to the policies. However, for illustration purposes we keep the example as simple as possible. When running Prover9 on $P_G \cup P_C$ we obtain a proof, which looks like follows:

```
3 (all A (user(A) & -superuser(A) -> -add_user(A)))
                  # label(non_clause).  [assumption].
4 (all A (admin(A) -> user(A))) # label(non_clause).
                  [assumption].
5 (all A (admin(A) -> add_user(A))) # label(non_clause).
                   [assumption].
6 -user(A) | superuser(A) | -add_user(A).  [clausify(3)].
9 -superuser(julia).  [assumption].
10 admin(julia).  [assumption].
11 -admin(A) | user(A).  [clausify(4)].
12 -admin(A) | add_user(A).  [clausify(5)].
13 -user(julia) | -add_user(julia).  [resolve(9,a,6,b)].
15 user(julia).  [resolve(10,a,11,a)].
16 -add_user(julia).  [resolve(15,a,13,a)].
17 add_user(julia).  [resolve(10,a,12,a)].
18 $F.  [resolve(16,a,17,a)].
```

Hence, we see that there is a contradiction, which have to be resolved. In this case eliminating the contradiction is easy. This can be done by adding `& -admin(X)` to the rule stating that ordinary users are not allowed to add new users. However, this would change the general policy and thus limit reuse. Therefore, we need an approach that allows us to change the policy on the fly when bringing together the general and the custom policy.

3 Consistency Handling

In the previous section we discussed the underlying problem of handling consistencies in case of general and custom policies used for implementing a test oracle in the domain of security testing. In this section, we discuss how to handle cases where the general policy is in contradiction with the custom policy. Such cases require actions for eliminating the root causes of the contradiction. As already mentioned the approach introduced in this section for retaining consistency is based on the artificial intelligence technique model-based diagnosis [4, 9, 6]. There assumptions about the health state of components are used to eliminate inconsistencies arising when comparing the expected with the observed behavior. In the case of our application domain, i.e., eliminating inconsistencies that are caused by the rules of a policy, we have to add assumptions about the validity of the rules.

In the following, we discuss the approach using the example of Section 2. The approach is based on the following underlying assumptions:

- We assume that both the general policy and the custom policy is consistent.
- We prefer custom policy rules over general policy rules. As a consequence we search for rules of the general policy that can be removed in order to get rid of inconsistencies.
- We prefer any solutions that are smaller in terms of cardinality than others. Here the underlying rational is that we want to remove as little information as possible stored in any knowledge base.

The first step of our approach is to convert the first-order logic policy rules into a propositional form. We do this by replacing each quantified variable with a constant. The set of constants to be considered are the constants used somewhere in the policy rules. In our running example we only have the two constants `julia` and `adam`. Note that this transformation does not change the semantics in our case because all the rules are specific to a certain instantiation. The constants represent such an instantiation. Therefore, there is no negative impact on the semantics. For our example we would obtain the following set of rules after conversion to propositional logic:

```
(superuser(adam) -> user(adam)).
(superuser(julia) -> user(julia)).
(superuser(adam) -> add_user(adam)).
(superuser(julia) -> add_user(julia)).
((user(adam) & -superuser(adam)) -> -add_user(adam)).
((user(julia) & -superuser(julia)) -> -add_user(julia)).
admin(adam) -> user(adam).
admin(julia) -> user(julia).
admin(adam) -> add_user(adam).
admin(julia) -> add_user(julia).
admin(julia).
-superuser(julia).
user(adam).
```

In the next step, we introduce the predicate $\texttt{ab}_i$ to indicate that a certain rule i is valid or not. If a rule is valid the predicate is $false$, and otherwise $true$. For example, `(superuser(adam) -> user(adam))` might be a valid rule in the overall policy or not. In case the rule is not valid, we know that the its corresponding predicate has to be $true$. Hence, we modify the rule to:

```
ab1 | (superuser(adam) -> user(adam)).
```

We do this for all general rules only because of our assumption to prefer custom policy rules and obtain the following set of rules:

```
ab1 | (superuser(adam) -> user(adam)).
ab2 | (superuser(julia) -> user(julia)).
ab3 | (superuser(adam) -> add_user(adam)).
ab4 | (superuser(julia) -> add_user(julia)).
ab5 | ((user(adam) & -superuser(adam)) -> -add_user(adam)).
ab6 | ((user(julia) & -superuser(julia)) -> -add_user(julia)).
admin(adam) -> user(adam).
admin(julia) -> user(julia).
admin(adam) -> add_user(adam).
admin(julia) -> add_user(julia).
admin(julia).
-superuser(julia).
user(adam).
```

The third step comprises searching for truth assignments to $\texttt{ab}_i$ such that no contradiction arises. The truth assignment to all $\texttt{ab}_i$ that leads to no contradiction is an explanation and all rules where $\texttt{ab}_i$ is set to true can be eliminated

from the general policy. Hence, the union of the rules from the general policy and the custom policy is guaranteed to be consistent by construction. Formally, we define explanations as follows:

Definition 1. *Let K be the propositional knowledge base obtained from the policy $P = P_G \cup P_C$ and AB the set of abnormal predicates $\mathtt{ab}_i$ used in K. A set $E \subseteq AB$ is an* explanation *if and only if setting all elements of E to true and the rest of the predicates to false causes K to be satisfiable, i.e., $K \cup \{\mathtt{ab}_i | \mathtt{ab}_i \in E\} \cup \{-\mathtt{ab}_i | \mathtt{ab}_i \in AB \setminus E\} \not\models \bot$.*

It is worth noting that this definition of explanations corresponds to the definition of diagnosis in the context of model-based diagnosis. We refer the interested reader to Reiter [9] for more information. In contrast to Reiter we do not need to define explanations as minimal explanations. Instead we say that an explanation E is minimal, if there exists no other explanation E' that is a subset of E.

For example, in our running example `ab6` is an explanation, which can be easily verified by calling Prover9 with the propositional knowledge base including the logical rule `-ab1 & -ab2 & -ab3 & -ab4 & -ab5 & ab6`. In this case Prover9 does not return a proof. Therefore, the policy is consistent and can be used as test oracle. Similarly, we are able to check that `ab5`, `ab6` together is also an explanation. However, this explanation is not minimal accordingly to our definitions.

Computing explanations can be done by checking all subsets of AB using a theorem prover. Such an algorithm of course would be infeasible especially for larger knowledge bases. In order to improve computing explanations we are able to use the fact that we are interested in smaller explanations. Hence, we might check smaller subsets first. We refer the interested reader to Reiter [9] and Greiner [6] for an algorithm that is based on this idea.

In the following we outline a novel algorithm **COMPEX** (Algorithm 1) that computes one single explanation for a knowledge base and the set of possible assumptions. **COMPEX** basically comprises two parts. In Line 1 to Line 8 an explanation is computed. This is done by calling the theorem prover Prover9 and extracting all $\mathtt{ab}_i$ predicates that are used by Prover9 to derive a contradiction. These predicates are added to the explanation E. The second part (Line 9 to 17) is for minimizing E. This is done by removing element by element from E. If we still obtain a consistent explanation when removing the element, the element is not part of the explanation. Otherwise, the element is important and, therefore, added again to the explanation (see Line 15). Although, **COMPEX** makes us of Prover9, the theorem prover can be replaced. The only requirement here is that a theorem prover returns information regarding the `ab` predicates used to derive a contradiction and the empty set if no contradiction can be derived.

It is worth noting that **COMPEX** does not necessarily return a minimal explanation. However, it ensures that consistency is retained. Moreover, **COMPEX** is efficient and always terminates with an explanation accordingly to Definition 1. The correctness of the output is ensured by construction because of the

Algorithm 1 COMPEX(K, AB)

Input: A propositional knowledge base K and a set of abnormal predicates $AB = \{\texttt{ab1}, \ldots, \texttt{ab}_n\}$
Output: An explanation

```
1: E = ∅
2: CS = {−ab_i | ab_i ∈ AB}
3: Pr = Prover9(K ∪ CS)
4: while Pr ≠ ∅ do
5:    Add all ab_i predicates used in Pr for deriving a contradiction to E.
6:    CS = {−ab_i | ab_i ∈ AB \ E} ∪ {ab_i | ab_i ∈ E}
7:    Pr = Prover9(K ∪ CS)
8: end while
9: for i = 1 to |E| do
10:    Let e be the i-th element of E
11:    E = E \ {e}
12:    CS = {−ab_i | ab_i ∈ AB \ E} ∪ {ab_i | ab_i ∈ E}
13:    Pr = Prover9(K ∪ CS)
14:    if Pr ≠ ∅ then
15:       E = E ∪ {e}
16:    end if
17: end for
18: return E
```

consistency checks performed by the theorem prover. Termination is guaranteed because in every theorem prover call more `ab` predicates are added. This process has to stop because AB is finite and assuming that consistency can be retained using the `ab` predicates introduced.

Computational complexity of **COMPEX** is in the worst case of order $O(|AB| \cdot TPC)$ where TPC represents the worst case execution time for calling the theorem prover. Because of the fact that this time is in the worst case exponential for propositional logic, TPC is the main source of computational complexity. In order to give an indication of feasibility of **COMPEX** we carried out two case studies. In the first one, we checked the time for computing the explanations using our running example and other knowledge base comprising 39 clauses and 26 assumptions. For the running example time was less than 0.00 seconds. For the other knowledge base we required 4 theorem prover calls, each requiring 0.01 seconds at the maximum. Hence, again computing explanations was possible within a fraction of a second. The purpose of the second case study was to show the performance in one extremal case. For this purpose we borrowed the problem of computing all diagnoses of an inverter chain, where each inverter i is modeled as $\texttt{ab}_i \mid \texttt{(i}_{i-1} \texttt{ <-> i}_i \texttt{)}$. For a chain of inverter **COMPEX** cannot reduce the number of `ab`'s in the first stage and thus has to call the theorem prover $n + 2$ times with n being the number of inverters. Note that for this example we set $\texttt{i}_0$ to be true and the final output to a value, which is not the expected one.

When using Prover9 and the inverter chain examples ranging from n equals 50 to 500 we obtained the time (in log scale) depicted in Figure 1 required for carrying out all necessary theorem prover calls. We see that the time complexity for this example is polynomial. The equation $t = 5 \cdot 10^{-7} \cdot n^{-3.487}$ fits the given curve with a coefficient of determination value R^2 of 0.99861. When assuming that we do not need to compute the reasons for inconsistency interactively, even larger knowledge bases of several hundreds of clauses can be handled within less

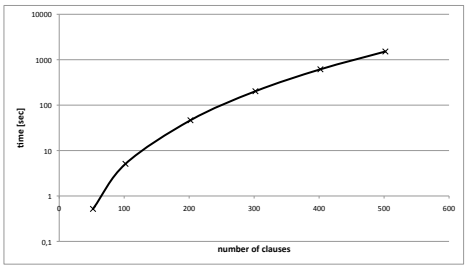

Fig. 1. Time for computing all theorem prover calls for the inverter chain example

than half an hour, which seems to be feasible for our working domain. Moreover, other theorem provers or constraint solver might lead to even better figures.

In order to finalize this section we briefly summarize the method of constructing the overall policy to be used as test oracle. This is done by first constructing a propositional policy from the general and the custom policy where quantified variables are instantiated using constants of the policies. All propositional rules that belong to a rule of the general policies are extended with a disjunction of a corresponding abnormal predicate. After all calls to **COMPEX** those rules are eliminated where their corresponding abnormal predicate is element of the returned solution. The remaining rules are ensured to be consistent and form the knowledge-base of the test oracle.

4 Related Research and Conclusions

Retaining consistencies in knowledge bases has been considered as important research question for a long time. Shapiro [10] dealt with debugging Prolog programs in case of incorrect behavior. Later Console et al. [3] provided a model-based debugging approach with some improvements. Bond and Pagurek [1] improved the work of Console and colleagues. Based on theses previous work Felfernig et al. [5] introduced an approach for diagnosis of knowledge bases in the domain of configurations. This work is close to our work. However, there are several differences. The algorithm presented in this work is adapted towards the desired application domain, i.e., security testing, and thus does not need to provide all possible solutions. Moreover, we are not interested in finding the most general solution at the level of first order logical sentences. Therefore, finding solutions becomes decidable again. Moreover, when restricting to one solution, the computation also becomes tractable when considering the efficiency of today's theorem provers and SAT solvers.

In this paper we briefly discussed a framework for testing systems with the focus on security properties. For this purpose we assumed that we have general knowledge that captures security issues like stating that ordinary users are not allowed to add new users to a system. In order to test a specific application the

general knowledge is accompanied with security properties that arise from the requirements and specification of the system itself. This separation of knowledge bears the advantage of increased reuse but at the same time causes challenges due to inconsistencies between the general and the custom knowledge.

In order to solve the inconsistency challenge we introduced a method for delivering the causes of inconsistencies. This causes have to be removed in order to retain consistency. The presented novel algorithm is specifically tailored for the application domain and is based on model-based diagnosis from which we took the basic idea. Future research include the formalization of available security knowledge in order to carry out further case studies and to provide an empirical study with the objective to prove the applicability of the approach in the security domain.

Acknowledgments. This research has received funding from the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and the Austrian COMET Program (FFG).

References

1. Bond, G.W.: Logic Programs for Consistency-Based Diagnosis. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada (1994)
2. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. **23**(7) (July 1997) 437–444
3. Console, L., Friedrich, G., Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In: International Joint Conference on Artificial Intelligence (IJCAI), Chambery (August 1993) 1494–1499
4. Davis, R.: Diagnostic reasoning based on structure and behavior. Artificial Intelligence **24** (1984) 347–410
5. Felferning, A., Friedrich, G., Jannach, D., Stumptner, M.: Consistency based diagnosis of configuration knowledge bases. Artificial Intelligence **152**(2) (2004) 213–234
6. Greiner, R., Smith, B.A., Wilkerson, R.W.: A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence **41**(1) (1989) 79–88
7. Hoglund, G., McGraw, G.: Exploiting Software: How to Break Code. Addison-Wesley (2004) ISBN: 0-201-78695-8.
8. McCune, W.: Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/` (2005–2010)
9. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (1987) 57–95
10. Shapiro, E.: Algorithmic Program Debugging. MIT Press, Cambridge, Massachusetts (1983)