

An advanced approach for modeling and detecting software vulnerabilities

Nahid Shahmehri^{a,*}, Amel Mammari^b, Edgardo Montes de Oca^c, David Byers^a, Ana Cavalli^b, Shanai Ardi^a, Willy Jimenez^b

^a Department of Computer and Information Science, Linköping University, SE-58183 Linköping, Sweden

^b Télécom SudParis, 9 rue Charles Fourier, 91011 Evry Cedex, France

^c Montimage, 39 rue Bobillot, Paris 75013, France

ARTICLE INFO

Article history:

Received 16 December 2010

Received in revised form 20 March 2012

Accepted 21 March 2012

Available online 30 March 2012

Keywords:

Software security

Security modelling

Secure software engineering

Automatic testing

Dynamic analysis

ABSTRACT

Context: Passive testing is a technique in which traces collected from the execution of a system under test are examined for evidence of flaws in the system.

Objective: In this paper we present a method for detecting the presence of security vulnerabilities by detecting evidence of their causes in execution traces. This is a new approach to security vulnerability detection.

Method: Our method uses formal models of vulnerability causes, known as *security goal models* and *vulnerability detection conditions* (VDCs). The former are used to identify the causes of vulnerabilities and model their dependencies, and the latter to give a formal interpretation that is suitable for vulnerability detection using passive testing techniques. We have implemented modeling tools for security goal models and vulnerability detection conditions, as well as *TestInv-Code*, a tool that checks execution traces of compiled programs for evidence of VDCs.

Results: We present the full definitions of security goal models and vulnerability detection conditions, as well as structured methods for creating both. We describe the design and implementation of *TestInv-Code*. Finally we show results obtained from running *TestInv-Code* to detect typical vulnerabilities in several open source projects. By testing versions with known vulnerabilities, we can quantify the effectiveness of the approach.

Conclusion: Although the current implementation has some limitations, passive testing for vulnerability detection works well, and using models as the basis for testing ensures that users of the testing tool can easily extend it to handle new vulnerabilities.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The presence of vulnerabilities – security-related flaws – in software has become a major concern in the software industry. Although efforts are being made to reduce security vulnerabilities in software, including advances in development processes and tools, the number of vulnerabilities and the number of computer security incidents that result from exploiting them remains very high [1].

In this paper we present a novel model-based approach to detecting evidence of vulnerabilities in traces from running software, a technique known as *passive testing*. Passive testing has proven to be very effective for detecting faults. An application of this approach to the WAP protocol is given by Bayse et al. [2], who

show how to use passive testing for conformance testing of WAP protocol implementations. Our approach differs from conventional vulnerability testing methods in that it uses vulnerability models that developers can inspect (and create) as the basis for detection.

Today, there are a large number of techniques and tools used in the software industry to improve software quality, from formal verification and validation to static and dynamic code analyzers and testing tools [3,4]. These tools are very focused on specific types of vulnerabilities, such as looking for insecure library functions, bounds checking errors, and various types of buffer overflows [5–7]. However, many suffer from several disadvantages, including:

- It is often difficult for users of the tools to know which vulnerabilities any given tool actually deals with.
- Users have no assurance that their tools will be kept up-to-date with changing risks and threats, and have little ability to keep the tools up-to-date without relying on the tool vendor.

Our approach addresses these issues: by using models as the basis for detection, developers can easily see exactly what the tool

* Corresponding author.

E-mail addresses: nahid.shahmehri@liu.se (N. Shahmehri), amel.mammari@it-sudparis.eu (A. Mammari), edgardo.montesdeoca@montimage.com (E. Montes de Oca), david.byers@liu.se (D. Byers), ana.cavalli@it-sudparis.eu (A. Cavalli), shanai.ardi@liu.se (S. Ardi), willy.jimenez@it-sudparis.eu (W. Jimenez).

detects, and the tool can be extended simply by adding new models, as new vulnerabilities are discovered. Creating new models requires some security expertise, particularly if they contain entirely new subgoals, but the process is easy to learn for experienced developers.

The detection models are based on security goal models (SGMs) [8], that model the causes of vulnerabilities. SGMs are an extension of vulnerability cause graphs (VCGs) [9], developed at Linköping University. VCGs were developed for software process improvement; SGMs add flexibility and expressive power that is necessary in applications such as passive testing. An SGM can show the potential causes of a vulnerability, as well as how the causes are related to each other. SGMs can be used to determine how to avoid introducing vulnerabilities, and they facilitate communication between the different stakeholders in software development (this was a key idea behind the SHIELDS project [10]). Our approach to passive testing uses SGMs as the basis for detecting evidence of vulnerabilities in execution traces.

Although the interpretation of the structure of an SGM is well defined, individual causes are described using natural language. As a result, SGMs alone are not suitable for use by automatic tools for vulnerability detection. We have developed a formal language, called *vulnerability detection condition* (VDC), that allows us to use SGMs for vulnerability detection. This is accomplished by defining each cause with a logical predicate, then composing the individual predicates into a VDC based on the transformation of the SGM into scenarios. The result is a precise and unambiguous definition of each cause, and by extension the entire SGM.

By basing VDCs on SGMs, we tie testing to other security-related activities that are supported by SGMs, such as manual inspection, static analysis and process improvement [11]. Furthermore, by using SGMs, the effort spent creating one VDC is trivially reused when VDCs are created for vulnerabilities with similar causes.

Known *kinds* of vulnerabilities account for nearly all reported software security problems, so the ability to detect them is a powerful tool for preventing security vulnerabilities during development. Since SGMs and VDCs are abstractions of concrete vulnerabilities, the model developed for a specific vulnerability can be used to detect similar vulnerabilities.

Finally, we introduce the *TestInv-Code* tool, developed by Montimage, that uses passive testing techniques and VDCs to detect vulnerabilities in C programs. We use the *TestInv-Code* tool to evaluate the effectiveness of the approach, detecting similar vulnerabilities in several different programs.

In summary, the main contributions in this paper are the following:

- A complete definition of the security goal model language, as well as a method for creating security goal models for vulnerabilities (see Sections 2 and 3).
- A formal approach to generate vulnerability detection conditions (VDCs) to express security goal models in a rigorous way without ambiguity (see Sections 4 and 5).
- An automatic method, based on VDCs, and a tool *TestInv-Code*, for automatically detecting vulnerabilities (see Sections 5 and 6).
- An evaluation of the proposed approach on several vulnerabilities and programs written in the C language. The vulnerabilities to be detected have been modeled by an SGM for which a VDC has been generated (see Section 7).

2. Security goal models

A security goal model (SGM) models how a given goal can be achieved. The language was developed as a more expressive replacement for vulnerability cause graphs [12] (used to model

the causes of vulnerabilities), security activity graphs [13] (used to model the alternatives for performing security-related activities), security goal indicator trees [14] (used to model the process of goal-driven inspection) and attack trees [15] (used to model how to perform attacks). SGMs provide richer relationships between model elements, a key property when being used for automated applications such as passive testing.

In the context of an SGM, a *goal* is anything that affects security or affects some other goal; it is not necessarily something desirable. Typical examples are vulnerabilities, security functionality, security-related software development activities, and attacks. An SGM is a directed acyclic graph. Vertices represent subgoals (specifically, vertices *refer* to subgoals, and multiple vertices may refer to the same subgoal); solid edges represent dependencies between subgoals; and dashed edges can be thought of as modeling information flow.

This section is intended to give a flavor of the models and what they can express. The full syntax and semantics of SGMs is presented in Section 3.

Fig. 1 shows an SGM that models a successful attack on an online software distribution service. The root of the SGM is *replace software* (a successful attack), and the remaining vertices are subgoals. Those drawn with angled sides are themselves modeled using SGMs. A subgoal can be achieved only if its predecessors are achieved: whether all predecessors, or just one, are required is indicated by *and* and *or*, respectively. The dashed edges at the top of the graph can be thought of as representing information flow between subgoals: in this model the person influenced in *influence person* is either the admin from *identify admin* or the developer from *identify developer*, depending on which of these subgoals is achieved.

The model in Fig. 1 shows numerous ways in which the attack *replace software* can be performed, such as getting a developer to replace the software, after identifying and influencing them (perhaps through threats or bribes); or performing an unauthorized upload after successfully getting access to a developer account by stealing the password database through an SQL injection attack.

Fig. 2 shows an SGM for a buffer overflow vulnerability, including an SGM for one of the subgoals. Here, the goal is the presence of the vulnerability, and the elements of the model are potential (possibly partial) causes of the vulnerability. This is the type of SGM we want to use with passive testing. A new feature in this example is the subgoal drawn with a black fill, *use adaptive buffers*. This is a *counteracting subgoal*, i.e. a subgoal that counteracts the overall goal. Again, the dashed lines can be thought of as information flow.

The model in Fig. 2 indicates two ways in which the vulnerability can be caused. In both cases we have a program that does not *use adaptive buffers* and accepts *data read from user*. One way the vulnerability is caused is having the *data read from user* used in an *unchecked integer arithmetic*, the result of which is used as the size parameter in an *unsafe use of malloc/calloc*. The other way the vulnerability can be caused is to copy data within a loop, without a range check, when the loop is controlled by data read from the user.

3. The SGM language

The definition of the SGM language consists of a syntactic domain and abstract syntax (elements that make up a model and how they can be combined), a visual representation (the symbols we use in the models), and a semantic transformation from the syntactic domain to a semantic domain. For a complete treatment of these terms as applied to modeling, see ‘Meaningful Modeling: What’s the Semantics of “Semantics”’ [16].

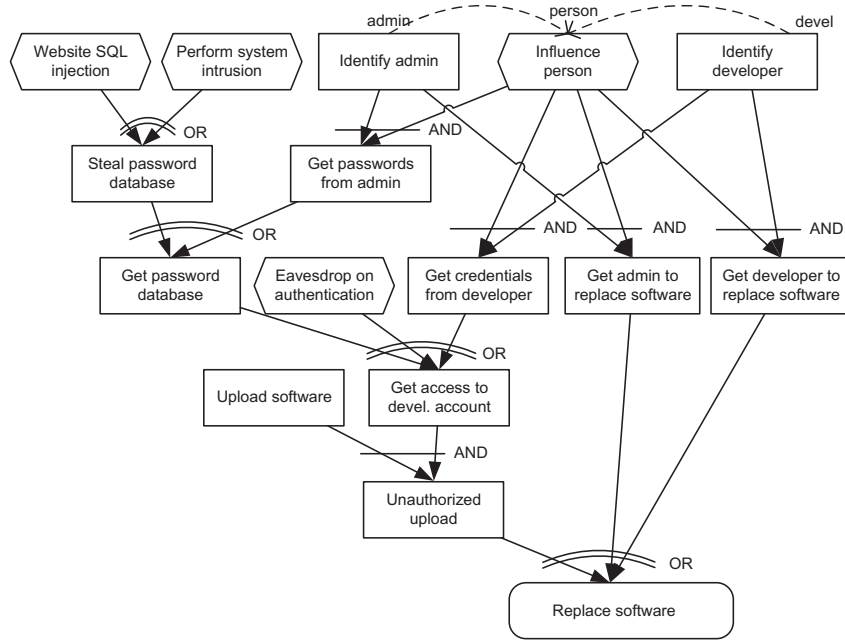


Fig. 1. Security goal model for the attack *replace software*.

3.1. Syntax

Fig. 3 shows the abstract syntax of the SGM language in a UML class diagram. Elements with italicized names are abstract and cannot appear directly in models. All generalizations are disjoint (members of a general class can be members of only one of its specializations) and complete (no specializations other than those shown exist) unless otherwise specified (this diagram is part of the larger SHIELDS metamodel [17], which is why some generalizations are incomplete and why all elements are specializations of *Resource*).

The **Root** is a vertex that is reachable through dependence edges from all subgoal elements. The root cannot have any successors through dependence edges. Returning to the examples in Section 2, the roots are the vertices at the bottom of each graph.

A **Subgoal** vertex represents a goal that contributes to (*contributing subgoal*) or counteracts (*counteracting subgoal*) the overall goal that the security goal model models. Every subgoal vertex must be associated with exactly one subgoal (subgoal vertices refer to subgoals, so multiple vertices can refer to the same subgoal). A subgoal vertex must have at most one predecessor (either an operator or a subgoal) and at least one successor through dependence edges. Apart from the roots, all vertices in Figs. 1 and 2 are subgoals. Those filled with black are counteracting; the others are contributing.

A **DependenceEdge** is a directed edge that represents any kind of dependence between subgoals and/or operators. An edge from A to B indicates that in order to fulfill the overall goal, both A and B must be fulfilled (assuming they are not counteracting). The reason for a dependence edge can be specified using stereotypes. All straight, solid edges in the examples in Section 2 are dependence edges.

Operators represent logical combinations of dependencies, either **And** or **Or**. Every operator must have at least one predecessor through a dependence edge (although two or more is typical) and exactly one successor through a dependence edge. Operators are visible as straight or curved lines (together with the words *and* or *or*) across the dependence edges of the examples in Section 2.

An **Annotation** is an arbitrary comment. Annotations may be associated with other model elements through **AnnotationEdges**. None of the examples in Section 2 contains annotations.

A **Stereotype** is an annotation on an edge, usually a dependence edge, used to explain why the edge exists. While the reason for including an edge does not affect whether the overall goal is fulfilled or not, it can make the graph easier to understand. The SGM in Fig. 2 demonstrates the stereotypes *uses* and *prerequisite*. Table 1 lists the stereotypes we have identified to date.

Subgoals can have one or more **InformationPorts**. Each port is either an **InputPort**, which represents some information used in the subgoal, or an **OutputPort**, which represents information produced by the subgoal. The *unchecked integer arithmetic* subgoal in Fig. 2 shows how information ports can be used: the subgoal has an input port representing operands and an output port representing the result.

Information ports can be connected using **InformationEdges**, which are used to add constraints to individual subgoals. For example, the edge from *unchecked integer arithmetic* to *unsafe use of malloc/calloc* in Fig. 2 signifies that the vulnerability may occur when the result of unchecked integer arithmetic is used in an unsafe call to `malloc`. The vulnerability does not occur if both unchecked arithmetic and unsafe calls to `malloc` exist, but are unrelated.

An information edge is directed and may originate at any port, but must terminate at an input port. An edge from A to B signifies that the information at port B is the same as the information at port A , provided that A and its dependencies are fulfilled.

The curved, dashed edges of the examples in Section 2 are information edges; the names of the information ports they are connected to are also shown.

Input and Output vertices link the ports of subgoals to their models. If a subgoal E is modeled by SGM G , then G contains one input vertex for each input port of E , and one output vertex for each output port of E . Every input vertex has one output port and every output vertex has one input port. The security goal model for *unsafe use of malloc/calloc* in Fig. 2 contains one input port at the top, labeled *size*, and an output port at the bottom, labeled *buffer*.

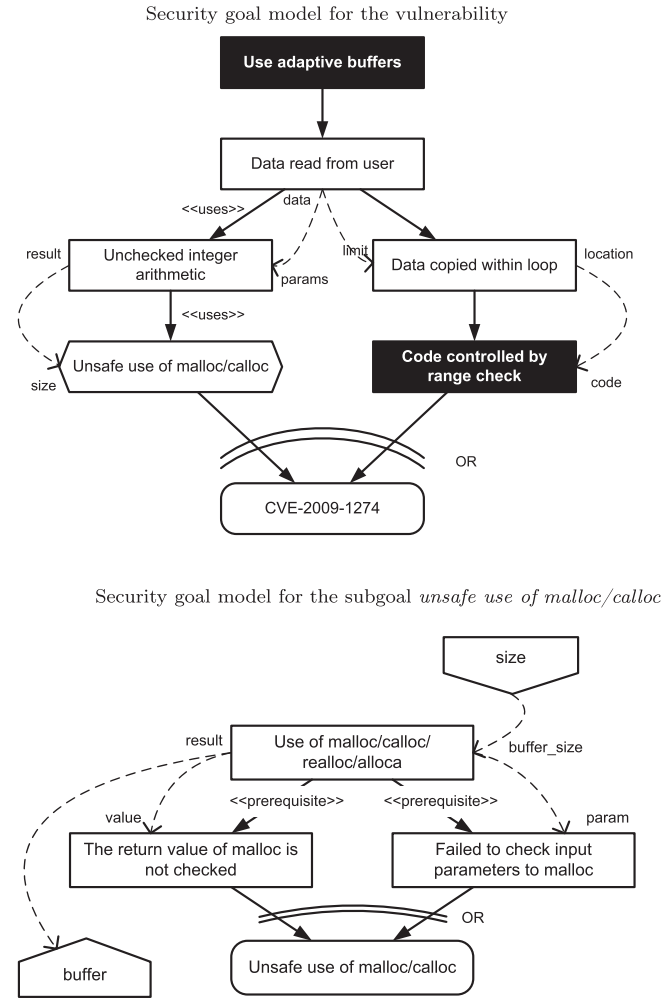


Fig. 2. Security goal model for the vulnerability CVE-2009-1274.

Model is the supertype of all models, and **ModelElement** is the supertype of all model elements. These are abstract, and thus never instantiated directly. **CoreElement** represents atoms of security knowledge, such as subgoals. It is through this supertype that models can be related to each other: models are representations of core elements, and model elements such as subgoal vertices refer to core elements.

Elements **SGMElement**, **Vertex**, and **Edge** are abstract, and thus never instantiated directly. The **SGM** element represents an entire SGM.

3.2. Visual representation

Elements in the syntactic domain are visually represented as shown in Table 2. Subgoals and dependence edges were chosen to be similar to corresponding elements in attack trees, vulnerability cause graphs, security activity graphs, and security goal indicator trees. Operations are reminiscent of Schneier's attack tree notation [15]. The design of the information edge was chosen to be visually distinct from dependence edges (bent, not straight, dashed, not solid). The dashed line reflects that information edges are less crucial to the model than dependence edges. The design of inputs and outputs are inspired by notation used in the Taverna workflow system [18].

3.3. Semantics

The full semantics of the SGM language is defined as two separate transformations: the first is a syntactic transformation that translates an object graph adhering to the abstract syntax into a 6-tuple, replacing the operators, input and output vertices with more general relations. The second is the semantic transformation that translates the 6-tuple into a set of scenarios, each of which describes a valid way to fulfill the modeled security goal (similar to Mauw and Oostdijk's formalization of attack trees [58]). The scenarios can then be interpreted in a manner appropriate to each individual application; in this paper they are used to create VDCs; in the SHIELDS project we translated them to Datalog [19], for use in static analysis [11].

3.4. Definitions

First we introduce some common notation: $\mathcal{P}(V)$ denotes the powerset of V ; $\mathcal{P}^+(V)$ denotes the set of non-empty subsets of V . $\mathcal{M}(V)$ is the set of sub-multisets of multiset V and $\mathcal{M}^+(V)$ is the set of non-empty sub-multisets of multiset V . Multisets are written $\{\{...\}\}$. The distributed product of sets of multisets is defined as $V \otimes W = \{v \uplus w | v \in V, w \in W\}$. The operator $\otimes_{i \in I}$ is the generalization of \otimes .

Given a relation $R \subseteq N \times \mathcal{M}^+(N')$, where $N' \subseteq N$, we say that n_2 is *reachable through* R from n_1 iff there exists an element (n_1, X) in R such that $n_2 \in X$ or n_2 is reachable through R from any element in X . We say that R is *acyclic* if there is no element n such that n is reachable through R from itself.

Table 3 lists the basic domains and functions used in the definition of the translations, as well as in the Datalog interpretation.

We use a definition of security goal models that eliminates operators, input vertices and output vertices. Operators are replaced by the \rightarrow_d relation, which maps a vertex n to a set whose elements are multisets of vertices that n depends on. The ports of all input and output vertices are associated with the SGM root instead.

Definition 1 (Security goal model, SGM). A security goal model T is a 6-tuple $(N, P, \text{node}, \rightarrow_i, \rightarrow_d, n_0)$, where N is a finite subset of **Node** such that $n_0 \in N$; P is a finite subset of **Port**; **node** is a function $\text{node}: P \rightarrow N$ denoting the node that a given port belongs to; \rightarrow_i is finite acyclic relation $\rightarrow_i \subseteq P \times P$ denoting the information edges in the SGM; \rightarrow_d is a finite acyclic relation $\rightarrow_d \subseteq N \times \mathcal{M}^+(N \setminus \{n_0\})$ denoting the dependencies in the SGM; and n_0 is the root of the SGM. The set of end nodes of the SGM T is defined as $E(T) = \{n \in N | \nexists x: n \rightarrow_d x\}$. All $n_i \in N \setminus \{n_0\}$ must be reachable through \rightarrow_d from n_0 .

3.5. Translation from graphical notation

Let G be an SGM in graphical notation. We define the helper sets V , W , X and helper function $D: \text{Node} \rightarrow \mathcal{P}(\mathcal{M}^+(\text{Node}))$:

$$\begin{aligned} V &= \{n \in G.\text{vertices} | \text{Types}(n) \cap \{\text{Input}, \text{Output}\} \neq \emptyset\} \\ W &= \{n \in G.\text{vertices} | \text{Subgoal} \in \text{Types}(n)\} \\ X &= \{e \in G.\text{edges} | e \in \text{IE}\} \end{aligned}$$

$$D(n) = \begin{cases} \{\{n\}\} & \text{if } n \in W \\ \bigcup_{p \in n.\text{dependsOn}} D(p) & \text{if } \text{Or} \in \text{Types}(n) \\ \bigotimes_{p \in n.\text{dependsOn}} D(p) & \text{if } \text{And} \in \text{Types}(n) \end{cases}$$

The set V is the set of input and output vertices. These are not included in the 6-tuple, but their ports will be associated with the root. The set W is the set of all subgoals and the set X is the

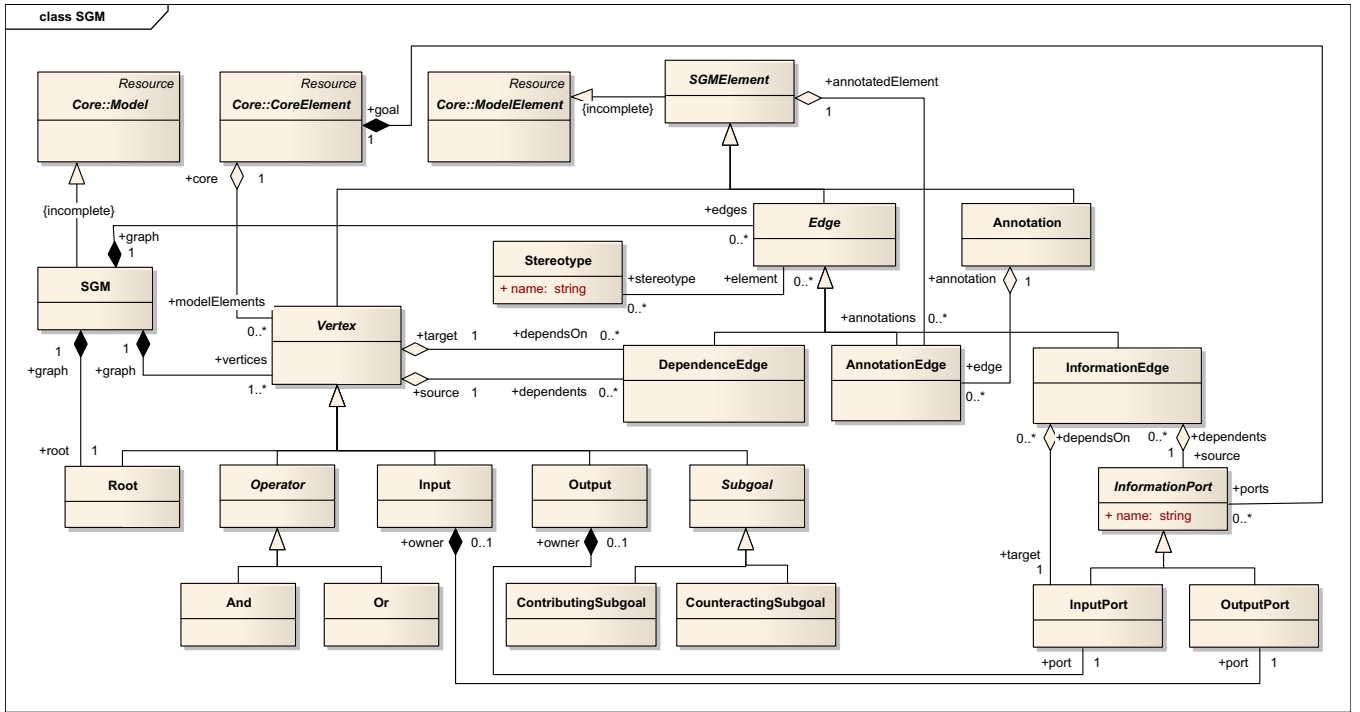


Fig. 3. Abstract syntax for the security goal model language.

Table 1
Stereotypes identified in SGMs to date.

Name	Explanation
Causes	A subgoal is the direct consequence of another
Prerequisite	A subgoal cannot be fulfilled unless another has also been fulfilled. For example, the existence of a logging component is a prerequisite to messages being sent to the logging component
Uses	An operation uses the result of another. For example, a memory allocation may use the result of an unchecked multiplication

set of all information edges. Finally, the function D encodes the effects of operations, which are not included in the 6-tuple. We can now derive the SGM $T = (N, P, \text{node}, \rightarrow_i, \rightarrow_d, n_0)$ as follows:

$$\begin{aligned}
 N &= W \cup \{G.\text{root}\} \\
 P &= \{n.\text{port} \mid n \in V\} \cup \left(\bigcup_{n \in W} n.\text{ports} \right) \\
 \text{node} &= \{p \mapsto n \mid n \in W : p \in n.\text{ports}\} \cup \\
 &\quad \cup \{p \mapsto n_0 \mid \exists n \in V : p = n.\text{port}\} \\
 \rightarrow_i &= \{(s, d) \mid \exists e \in X : s = e.\text{source}, d = e.\text{target}\} \\
 \rightarrow_d &= \{(n, s) \mid n \in N, n.\text{dependsOn} \neq \emptyset, \\
 &\quad s \in D(n.\text{dependsOn})\} \\
 n_0 &= G.\text{root}
 \end{aligned}$$

The set N is the set containing all subgoals and the root. The set P is constructed by collecting all ports from both input and output vertices, and from subgoals. The node function is constructed by mapping each port to the subgoal it was associated with, and ports associated with input and output vertices to the root. The \rightarrow_i relation is essentially a copy of all information edges, with edges moved from input and output vertices to the root. The construction of the \rightarrow_d relation maps non-end nodes to their predecessors in the graph, with the effects of operations accounted for.

3.6. Transformation to scenario suites

The semantics of a security goal model is defined in terms of *scenarios* and *scenario suites*. A scenario is a set of subgoals together with the information edges that connect them. A scenario suite is a set of scenarios. The semantics of an SGM is the scenario suite that contains all scenarios that represent ways to fulfill the SGM.

Let T be an SGM $(N, P, \text{node}, \rightarrow_i, \rightarrow_d, n_0)$. The semantic transformation of T , $\mathcal{S}[\cdot] : \mathbf{SGM} \rightarrow \mathbf{Suite}$ is defined by:

$$\mathcal{S}[T] = \{(b, e) \mid b \in \mathcal{N}[n_0], e = I(b)\}$$




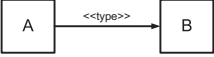
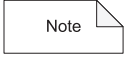





$$\mathcal{N}[n] = \begin{cases} \{\{n\}\} & \text{if } n \in E(T) \\ \{\{n\}\} \otimes \left(\bigcup_{n \rightarrow_d x, m \in X} \mathcal{N}[m] \right) & \text{if } n \notin E(T) \end{cases}$$

$$I(b) = \{(x, y) \in \rightarrow_i \mid \text{node}(x) \in b, \text{node}(y) \in b\}$$

The semantic transformation $\mathcal{S}[T]$ is valid if, and only if, $\bigcup_{(b, e) \in \mathcal{S}[T]} e = \rightarrow_i$ (i.e. every information edge is included in some scenario).

Table 2

Visual representation of security goal model elements.

Symbol	Element
	Vertex representing a subgoal that is not associated with a security goal model. <i>A</i> is a contributing subgoal; <i>B</i> is a counteracting subgoal
	Vertex representing a subgoal associated with a security goal model. <i>A</i> is a contributing subgoal; <i>B</i> is a counteracting subgoal
	Root
	Dependence edge with stereotype. <i>B</i> depends on <i>A</i> . The edge has stereotype <i>type</i>
	Note annotation
	Annotation edge; <i>A</i> is an annotation for <i>B</i>
	Operation <i>or</i> and operation <i>and</i> , with three operands each. Text labels are optional
	Information edge. There is an information edge connecting information port <i>src</i> of vertex <i>A</i> to information port <i>dst</i> of vertex <i>B</i>
	Port type indications: <i>src</i> is an output port and <i>dst</i> is an input port. These indications are optional
	Input (named <i>input</i>) and output (named <i>output</i>)

3.7. Example of the semantic transformation

Fig. 4 is the SGM shown in Fig. 2 (causation of vulnerability CVE-2009-1274) with shorter names. The semantic transformation consists of two steps: translation from the graphical notation followed by transformation to scenario suites.

Translation from the graphical notation yields the SGM $T = (N, P, \text{node}, \rightarrow_i, \rightarrow_d, n_0)$, where:

$$\begin{aligned}
 N &= \{A, B, C, D, E, F, G\} \\
 P &= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\} \\
 \text{node} &= \{p_1 \mapsto B, p_2 \mapsto C, p_3 \mapsto C, p_4 \mapsto D, \\
 &\quad p_5 \mapsto E, p_6 \mapsto E, p_7 \mapsto F\} \\
 \rightarrow_i &= \{(p_1, p_2), (p_3, p_4), (p_1, p_5), (p_6, p_7)\} \\
 \rightarrow_d &= \{(B, \{\{A\}\}), (C, \{\{B\}\}), (D, \{\{C\}\}), (E, \{\{B\}\}), \\
 &\quad (F, \{\{E\}\}), (G, \{\{D\}\}), (G, \{\{F\}\})\} \\
 n_0 &= G
 \end{aligned}$$

In this 6-tuple N contains the subgoals and the root, P is the set of all ports, node maps ports to the vertices they belong to, \rightarrow_i is the set of information edges, and \rightarrow_d the set of dependencies. Note the values

of $G \rightarrow_d n$: the *or* operation has resulted in two tuples involving G ; had there been an *and* operation instead, there would have been a single tuple, $(G, \{\{D, F\}\})$.

The next step transforms the 6-tuple T into a scenario suite. The fact that there are two tuples $(G, _)$ in \rightarrow_d will cause the scenario suite to contain two distinct scenarios:

$$\begin{aligned}
 S[T] &= \{(\{A, B, C, D, G\}), \{(p_1, p_2), (p_3, p_4)\}), \\
 &\quad (\{A, B, E, F, G\}), \{(p_1, p_5), (p_6, p_7)\})\}
 \end{aligned}$$

In other words, the goal modeled can be achieved in two different ways; in this case that means there are two ways to cause the modeled vulnerability. When performing passive testing, we will attempt to detect each scenario expressed by the SGM.

This example does not require the use of multisets. Multisets allow a given node to appear more than once in a scenario, but models, where this occurs are, in our experience, atypical. Fig. 5 shows a model with a single scenario, in which node A appears twice. For applications such as testing, a transformation that yielded a single A would have been adequate, but for other applications of the transformation that may not be the case (e.g. when used to detect redundancies in models).

Table 3

Domains used in the semantic transformation for SGMs.

Name	Description
SGM	The universe of all security goal models
Node	The universe of all SGM subgoals
Port	The universe of all SGM ports
Type	The universe of all type names (see Fig. 3)
$IE = Port \times Port$	The universe of all information edges
$Scenario = \mathcal{M}^+(Node) \times \mathcal{P}(IE)$	The universe of all scenarios
$Suite = \mathcal{P}(Scenario)$	The universe of all scenario suites
$Types : Node \rightarrow \mathcal{P}^+(Type)$	The types of a node

3.8. Creating security goal models

Security goal models that model vulnerabilities are created using a process similar to a typical root cause analysis. Starting with the root, subgoals are incrementally identified until a complete model has been created. The complete model is then subjected to a validation process to assure its quality. Ideally, SGMs would be created using a tool that allows the reuse of subgoals from other models – in our experience similar vulnerabilities will have similar subgoals. The complete procedure for creating SGMs is described in SHIELDS project deliverable D2.3 [17]; a tool for editing security goal models that supports reuse through the SHIELDS vulnerability repository is available from the SHIELDS website [10].

4. Passive testing for vulnerability detection

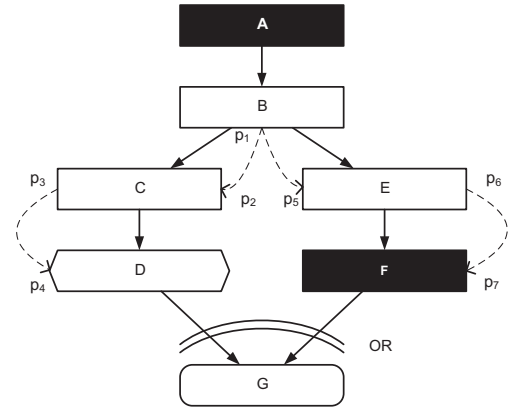
Passive testing is a testing method that detects faults by examining the traces of a software system without the need for specific test inputs [20]. In other words, passive testing is meant to detect faults in a system under test (SUT) by observing its behavior, or other observable characteristics, ideally without interfering at all with its normal operation. Basically, this testing technique consists of collecting traces produced by the SUT and trying to detect deviations or faults by comparing these traces to a formal model. Trace collection typically impacts performance, but does not require the SUT to be run with, e.g. special test vectors. The formal model can be a specification [21–23] of the underlying system or the properties [2,24,25] that the SUT must fulfill. For example, one approach to passive testing is to use a Finite State Machine (FSM) to model the expected behavior of a system. In this way it is possible to compare the execution traces to the FSM in order to detect faults in the implementation. Passive testing has been used in many different contexts, e.g. on an FSM model of a network, in network management to detect configuration provisioning and in a GSM-MAP protocol [26].

TestInv-Code, developed by Montimage, is a prototype passive testing tool that accepts formal vulnerability models written using VDCs. It detects vulnerabilities in an application by analyzing the traces of the code while it is executing. The traces used by *TestInv-Code* are the assembly code instructions that are being executed. These are produced by executing the program under the control of the *TestInv-Code* tool.

In order to use the *TestInv-Code* tool, the first step is defining the vulnerability causes that are of interest. Starting from the SGM and VDC models, a set of conditions that lead to a vulnerability are derived. These conditions are formally specified as regular expression-like rules.

Thus, passive testing using *TestInv-Code* proceeds along the following steps:

1. **Vulnerability modeling.** The vulnerability to be detected is modeled with an SGM.

**Fig. 4.** Security goal model for the causes of CVE-2009-1274 (simplified names).

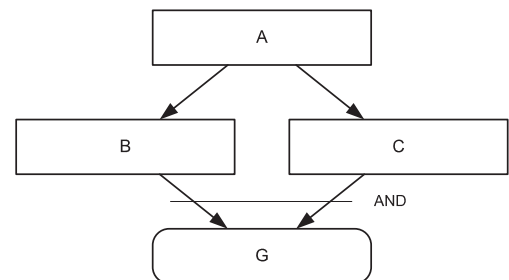
2. **Formal definition of causes.** A security expert associates a predicate with each cause. The predicates precisely define what the testing tool will look for in the execution traces. Note that each predicate needs to be implemented by the tool.
3. **Conversion of SGM to VDC.** A VDC for the entire vulnerability is constructed automatically from the SGM by using the structure of the SGM and the formal definitions of each individual cause and identifying the causality between each cause.
4. **Vulnerability checking.** Finally, *TestInv-Code* checks for evidence of the vulnerability during the execution of the program. Using the VDCs and the corresponding rules, it will analyze the execution traces to produce messages identifying the vulnerabilities found, if any, indicating, where they are located in the code.

It must be stressed that the first two steps are done manually, but only once per class of problem to be treated. The other steps are automated except for some conditions, where the procedure needs input from the user. User input is required when a condition cannot be automatically determined by the tool because the condition is such that it produces no evidence in the execution trace.

In Section 6 we show a few of the rules to illustrate how VDCs are used by *TestInv-Code* to detect vulnerabilities.

Fig. 6 depicts the passive testing architecture for vulnerability detection. As shown, the *TestInv-Code* tool takes as input:

1. **The VDCs.** The file containing the vulnerability causes formally specified using VDCs in an XML format [17,11].
2. **The executable.** The Executable Linked Format (ELF) file for the application that is to be tested. This file contains the binary code of the application. The tool will use this to execute the application and analyze its execution to detect any of the VDCs provided as input. Note that to determine the line of source code, where the vulnerability occurs, the file must include debug information (i.e. a symbolic debugging table). If this is

**Fig. 5.** Security goal model, where multisets are required in the semantic transformation.

not available, then the tool will only indicate the address of the instruction in the ELF file, where it detected the vulnerability.

5. Vulnerability detection conditions

An SGM can show how a software vulnerability can be caused. The different causes in the model provide information about possible problems in the code; such information can be used as the requirements in order to test software for vulnerability detection. Vulnerability detection conditions (VDCs) are then a formal interpretation of SGMs that model software vulnerabilities. The aim is to formally define the causes in the SGM in order to detect their occurrence in a piece of software.

Since causes are expressed in natural language and can represent anything, including conditions or events that an automatic tool cannot interpret (e.g. training of the programmer or business issues), it is possible to formally define and automatically detect only a subset of causes given by the SGM. Such causes are formalized using two predefined templates. A template is a table with specific and fixed fields conceived to systematically extract information required for software testing. Once filled, the templates are automatically processed to generate the VDCs as well as any other information useful for testing. A more formal definition of this concept follows.

Definition 2 (*Vulnerability detection condition*). Let Act be a set of action names, Var be a set of variables, and P be a set of predicates on $(Var \cup Act)$. A vulnerability detection condition VDC is of the form (square brackets denote an optional element):

$$VDC ::= a/P(Var, Act) | a[P(Var, Act)]; P'(Var, Act)$$

where a denotes an action, called a master action, that produces the vulnerability, $P(Var, Act)$ and $P'(Var, Act)$ represent any predicates on variables Var and actions Act .

A vulnerability detection condition $a/P(Var, Act)$ means that the master action a produces a vulnerability when it occurs under specific conditions denoted by predicate $P(Var, Act)$.

A vulnerability may also occur due to the action that follows the master action. That case is represented by

$$a[P(Var, Act)]; P'(Var, Act)$$

This means that the master action a used under the optional conditions $P(Var, Act)$ is followed by a statement whose execution satisfies $P'(Var, Act)$. Naturally, if action a is not followed by an action, the predicate $P'(Var, Act)$ is assumed to be true.

Intuitively, VDCs are composed of actions and conditions. An *action* denotes a particular point in a program, where a task or an instruction that modifies the value of a given object is executed. Some examples of actions are variable assignments, copying memory or opening a file. A *condition* denotes a particular state of a program defined by the value and the status of each variable. For a buffer, for instance, we can find out if it has been allocated or not.

More complex vulnerability detection conditions can be built inductively using the different logical operators according to the following definition.

Definition 3 (*General vulnerability detection conditions*). If VDC_1 and VDC_2 are vulnerability detection conditions, then $(VDC_1 \vee VDC_2)$ and $(VDC_1 \wedge VDC_2)$ are also vulnerability detection conditions.

5.1. VDC examples

In order to clarify the concept we present some examples of VDCs. First, consider that we want to define a vulnerability

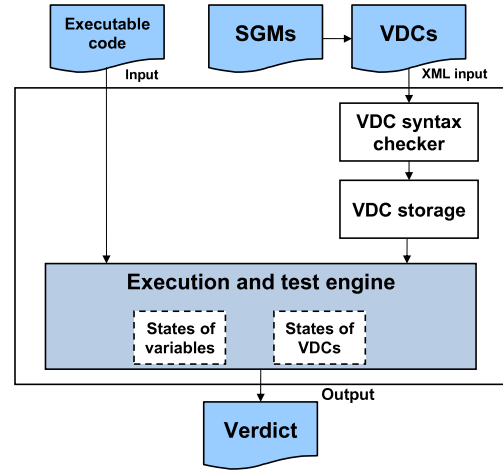


Fig. 6. Passive testing for vulnerability detection.

detection condition to detect if a certain value y is assigned to a memory variable x , but the memory space for x has not yet been allocated. We can define the VDC as follows:

$$VDC_1 = Assign(x, y) / IsNot.Allocated(x)$$

In the case of programming languages like C/C++, there are some functions that might lead to a vulnerability if they are applied on out-of-bounds arguments. The use of a tainted variable as an argument to a memory allocation function (e.g. `malloc`) is a well-known example of such a vulnerability, expressed by the vulnerability detection condition VDC_2 below. A variable is tainted if its value is obtained from a non-secure source. This value may be produced by reading from a file, getting input from a user or the network, etc. Note that a tainted value can be untainted during the execution of the program if it is checked to determine if it has an acceptable value.

$$VDC_2 = memoryAllocation(S) / tainted(S)$$

A good programming practice is to verify the return value from any allocation function. The following vulnerability detection condition VDC_3 detects the absence of such verification:

$$VDC_3 = (u := memoryAllocation(S)); notChecked(u, null)$$

5.2. Creating vulnerability detection conditions from SGMs

As we have mentioned previously, the aim of VDCs is to formally define the causes in the SGM in order to detect their occurrence in a piece of software. Therefore the process begins with the selection of the vulnerability of our interest with its corresponding SGM model. Then VDCs are created following four steps:

1. analyze the SGM that represents the vulnerability;
2. extract the testing information using templates;
3. automatically process the templates to obtain the VDCs; and
4. define the global VDC for the vulnerability.

These steps are discussed in detail in the following sections.

5.2.1. Analyze the SGM that represents the vulnerability

For a given SGM we need to identify all the possible scenarios that lead to the vulnerability in order to build testing scenarios. A testing scenario indicates that if the program under evaluation executes certain actions under some specific conditions then it contains the considered vulnerability.

These testing scenarios are defined through the semantic transformation of the SGM presented in Section 3. However, before applying the transformation we have to ensure that the SGM meets some specific requirements:

- If there is any subgoal modeled by a suitable SGM, replace it with its corresponding model.
- Discard the qualitative subgoals of the SGM and keep only the quantitative ones.
Qualitative subgoals cannot be checked or evaluated without human intervention. *Documentation is unclear* is an example of such a cause. Since our interest is automatic testing, we are concerned only with quantitative subgoals. A quantitative subgoal is directly related to the software code, so it can be automatically checked. An example is the use of `malloc` as memory allocation function.
- Replace counteracting subgoals with equivalent contributing subgoals. For testing we want to check the “bad” actions or conditions to determine whether the vulnerability is present or not.

The resulting graph is now adequate to obtain the VDCs. Nevertheless, in order to facilitate the scenario processing we use numbers to identify subgoals and information ports and then we apply the semantic transformation to obtain the testing scenarios.

5.2.2. Extract the testing information using templates

Once the scenarios are defined we have to collect all the possible details given by the subgoals and information ports and edges. The idea is to identify the variables, parameters, actions and conditions that contribute to the vulnerability. To do so we have created two templates, one corresponding to master actions and another to the conditions under which the master actions are executed. These templates are automatically processed to generate the VDCs.

In the SGM, every possible scenario must contain a single master action *Act_Master* that produces the related vulnerability; if some scenario is missing a master action, the SGM is not applicable to this testing method without further revision (also, the lack of a master action may indicate that the model is incomplete). All the other vertices of this path denote conditions $\{C_1, \dots, C_n\}$.

Among these conditions, a particular condition C_k may exist, called *missing condition*, which must be satisfied by an action following *Act_Master*. Let $\{P_1, \dots, P_k, \dots, P_n\}$ be the predicates describing conditions $\{C_1, \dots, C_k, \dots, C_n\}$. The formal vulnerability detection condition expressing this dangerous scenario is defined by:

$$Act_Master / (P_1 \wedge \dots \wedge P_{k-1} \wedge P_{k+1} \wedge \dots \wedge P_n); P_k$$

After the identification of master actions and conditions we take the corresponding template to analyze each subgoal. Also we have to consider information edges and ports since they provide information that clarifies the precise relationships between elements of the model, which is helpful from a testing perspective. For instance, information edges and ports can provide the names of variables and their relations with the different subgoals that cause the vulnerability.

For example, in Fig. 2 the subgoal *unchecked integer arithmetic* has an information edge to *unsafe use of malloc/calloc* to clarify that the use of functions *malloc/calloc* is unsafe if the result of an unchecked integer arithmetic expression is used as the size parameter.

The master action and condition templates are explained below.

Master Action Template This template is designed to capture all the information related to the master action of the SGM and possible input/output parameters. In some cases the name of some parameters and their relation with the subgoals can be given

directly by the information edges. The master action template with its corresponding items and a brief explanation of them are shown in Table 4.

From the template the master action expression is derived by combining some of the items according to the following general expressions:

- *function_name (input parameter)*: the master action is related to the execution of *function name* which receives *input parameter* as input.
- *function_name (output parameter, input parameter)* if the output parameter is given; the master action is related to the use of *function name* which receives *input parameter* as input to calculate the value of *output parameter*.

Condition Template The condition template is intended to describe the conditions under which the execution of the master action becomes dangerous, i.e., produces the modeled vulnerability. In some cases the name of some parameters and their relation with the subgoals can be given directly by the information edges. The condition template is described in Table 5.

The expression derived from condition template is written according to the formula:

- *Condition(name, condition_element)*. This indicates that the *condition* is given by *condition_element* acting on element *name*.

Let us remark that we specify the previous and next vertices for both master action and condition templates in order to be able to deduce scenarios from these templates independently from SGMs.

5.2.3. Automatically process the templates to obtain the VDCs for each scenario

In this step the information collected with the master action and condition templates are automatically processed to generate the expressions of the VDCs according to the corresponding testing scenario.

5.2.4. Define the global VDC for the vulnerability

The semantic transformation explained in Section 3 helps to find the scenario suite, a set of scenarios that show how many ways there are to cause the modeled vulnerability. From a testing perspective, we have to consider this scenario suite, which means we have to test all the scenarios in order to detect the considered vulnerability. Therefore, we define the global VDC representing the modeled vulnerability as the disjunction of all the vulnerability detection conditions for each scenario (VDC_i denotes the VDC associated with each path i):

$$VDC_1 \vee \dots \vee VDC_n$$

5.3. Example of VDC creation

The process of creating VDCs from SGMs is illustrated here through an example. Consider the SGM for CVE-2009-1274 in Fig. 2, which shows how a buffer overflow vulnerability is caused in the *xine* media player. Analyzing this model we observe the following features:

- There are six different subgoals. Two of them are counteracting subgoals: *use adaptive buffers* and *code controlled by range check*; while subgoal *unsafe use of malloc/calloc* is associated with an SGM.
- There are information edges that clarify the relationships between subgoals.

Table 4
Master action template.

Item	Description	Value
1. Vertex number	Number used to identify each vertex of the SGM	Integer
2. Previous vertex	This field indicates the number of the previous vertex in the SGM; it is duplicated from the SGM to make the template more self-contained	Integer
3. Next vertex	This field indicates the number of the next vertex/vertices in the SGM; it is duplicated from the SGM to make the template more self-contained	Integer (s)
4. Function name	Indicate the name of the master action function	Text (predefined)
5. Input parameter name	Indicate the name of the input parameter of the master action function	Free text
6. Input parameter type	Indicate the type of the input parameter of the master action function	Variable types
7. Variable that receives function result	Indicate the name of the variable that receives the result of the execution of the function considered	Free text
8. Type of the variable that receives function result	Indicate the type of the output parameter of the master action function	Variable types

Table 5
Condition template.

Item	Description	Value
1. Vertex number	Number used to identify each vertex of the SGM	Integer (s)
2. Previous vertex	This field indicates the number of the previous vertex of the SGM; it is duplicated from the SGM to make the template more self-contained	Integer
3. Next vertex	This field indicates the number of the next vertex of the SGM; it is duplicated from the SGM to make the template more self-contained	Integer
4. Search	Indicate the element considered in the node	Functions, variables, list
5. Name	Indicate the name of the element considered in the node	Free text or predefined (case of functions)
6. Type	Indicate the type of the element considered in the node	Predefined
7. Condition follows master action	Indicates if the current condition follows or not the execution of the master action	Yes or no
8. Condition	Condition expressed by the node	Reserved text
9. Condition element	Elements involved in the condition	Text

Thus, we have to transform this graph in order to create the VDCs: we replace the subgoal *unsafe use of malloc/calloc*

with its associated SGM, we also replace the counteracting subgoals with contributing ones and the resulting graph is shown in Fig. 7.

Applying the semantic transformation to the SGM of Fig. 7, the resulting scenario suite contains three scenarios that cause the modeled vulnerability (CVE-2009-1274):

$$S[T] = \{(\{1, 2, 3, 4, 5, 9\}, \{(p_1, p_2), (p_3, p_5), (p_4, p_7)\}), \\ (\{1, 2, 3, 4, 6, 9\}, \{(p_1, p_2), (p_3, p_5), (p_6, p_8)\}), \\ (\{1, 2, 7, 8, 9\}, \{(p_1, p_9), (p_{10}, p_{11})\})\}$$

Now a vulnerability detection condition has to be defined for each of these scenarios. The next part consists in identifying master actions. In our case, we can identify two different master actions that lead to the vulnerability, given by vertices 4 and 7.

Next we proceed with the analysis of vertex 4, which has three different information edges, two as output and one as input. The memory allocation in vertex 4 must use the variable `buffer_size` as parameter for the calculation of the size of the memory allocation. According to the information edge coming from vertex 3, the value of `buffer_size` is calculated as the result of unchecked integer arithmetic. Regarding the information edge to vertex 6, it indicates that the size parameter `buffer_size` is not properly validated before allocating the memory. Finally, the information edge to vertex 5 indicates that the result of the memory allocation function is not checked to verify that it is correct.

Summarizing, we have that variable `buffer_size` has to be considered at least in the templates for vertices 4, 5 and 6.

Vertex 7 is processed in a similar manner and the results of the analysis for both master actions are shown in Table 6.

The master action expressions are:

Alloc(buffer, buffer_size) and
CopyData(loop_counter, user_input).

The other nodes are analyzed with the condition template, considering the variables and functions indicated by master actions and information ports and edges. The process is shown with one vertex, number 2, which is about reading data from the user. Such a condition creates problems if the data is used in an integer arithmetic operation or if the data is copied inside a loop. We define a variable `user_input` in vertex 2, that holds the data provided by the user. The same variable has to appear in vertices 3 and 7 to keep the relation. Table 7 contains the results of the analysis for vertices 1, 2, and 3, 5, 6 and 8.

The predicates derived from the templates are listed in Table 8.

Finally, the vulnerability detection condition for scenario $(\{1, 2, 3, 4, 5, 9\}, \{(p_1, p_2), (p_3, p_5), (p_4, p_7)\})$ is given by the expression:

Table 6
Master action templates for CVE-2009-1274.

Item	Node	Node
1. Vertex number	4	7
2. Previous vertex	3	2
3. Next vertex	5,6	8
4. Function name	Alloc	CopyData
5. Input parameter name	Buffer_size	User_input, loop_counter
6. Input parameter type	Integer	String, integer
7. Variable that receives function result	Buffer	Buffer
8. Type of the variable that receives function result	Pointer	Pointer

$$VDC_1 = \text{Alloc}(\text{buffer}, \text{buffer_size}) / \left(\begin{array}{c} \text{Fixed}(\text{buffer}) \\ \wedge \\ \text{Result}(\text{buffer_size}, \text{user_input}) \\ \wedge \\ \text{Result}(\text{buffer_size}, \text{arithmetic}) \\ \text{Unchecked}(\text{buffer}, \text{null}) \end{array} \right);$$

This vulnerability detection condition expresses a potential vulnerability when the memory space for a non-adaptive buffer is allocated using the function `malloc` (or similar) whose size is calculated using data that is obtained from the user and the return value from memory allocation is not checked with respect to `null`.

In a similar way the VDCs for scenarios 2 and 3 are generated and the VDC for CVE-2009-1274 is given by the expression:

$$VDC = VDC_1 \vee VDC_2 \vee VDC_3$$

6. Using VDCs in TestInv-Code

TestInv-Code accepts VDCs encoded as XML documents from a VDC editor such as the one implemented in the SHIELDS project [17]. It is also capable of downloading published VDCs from the SHIELDS Security Vulnerabilities Repository Service [27].

In order to detect the presence of a VDC in an execution trace, it needs to be interpreted by the *TestInv-Code* tool. In a first version of the tool [28], conditions in the VDCs were translated into patterns of instructions that may appear in the execution trace. This worked correctly in particular cases, however, to be able to detect VDCs for executables produced from different code variations, by different compilers, another more generic technique has been developed, which is presented here.

In the latest version of the tool, abstract predicates such as *Alloc* and *CopyData* are implemented directly in the tool. *TestInv-Code* analyzes each instruction executed by the program and keeps track of the state of all the variables used by the program, including heap or stack memory addresses and registers. The states are, for instance, tainted or not, bounds checked or not, allocated or not,

etc. Essentially *TestInv-Code* is performing a limited form of reverse engineering to recognize the abstract predicates. Currently the user cannot easily add new predicates, but can define new VDCs using existing predicates. A future version of *TestInv-Code* will support adding new predicates through a plug-in mechanism.

To illustrate how a predicate is determined, we give some details on how *Unchecked(buffer, null)* works. In this case, *buffer* is space allocated on the heap using one of the normal memory allocation functions (e.g. `malloc` or `calloc`) and *Unchecked* means that the return value from the memory allocation has not been checked against, e.g. `NULL` before using the memory location. To be able to do this, *TestInv-Code* keeps track of all accumulators, registers and the instructions applied on them. For determining if a buffer is unchecked, the tool uses a function that will analyze each instruction and store the information necessary to allow it to set the state of the buffer to *checked* or *unchecked*. Table 9 shows examples of the code that the *Unchecked* predicate reacts to (C code and the corresponding assembly code, which is what *TestInv-Code* actually looks at) together with the actions taken by the tool.

The set of predicates implemented in *TestInv-Code* are designed to support our experiments and include *Unchecked*, *Unbounded*, *Tainted*, *Alloc* (only for heap memory), *Result*, *CopyData*, *Checked*, *Bounded*, *Untainted*, and *Freed*. New predicates are implemented as part of the tool, although a plug-in architecture will be available in future versions. All these functions examine the code being executed to determine when certain conditions hold or not, and they store this information so that it can be used when needed to verify the VDCs. Using these predicates, several VDC have been defined to test the method on different applications.

The tool is also able to detect when a system call is made, the checks that are made on variables or return values from function calls, when buffer allocations are made, etc., and this information can be used in the implementation of predicates. Thus it can verify all the conditions that are used in the VDCs and generate messages if the VDCs are satisfied.

For instance, the *Tainted* condition on a variable can be determined by detecting if the variable has been assigned values obtained by reading a file, a socket, entered by a user, etc. Taintedness can also be transmitted from one variable to another and a variable can become untainted if it is bounds checked by

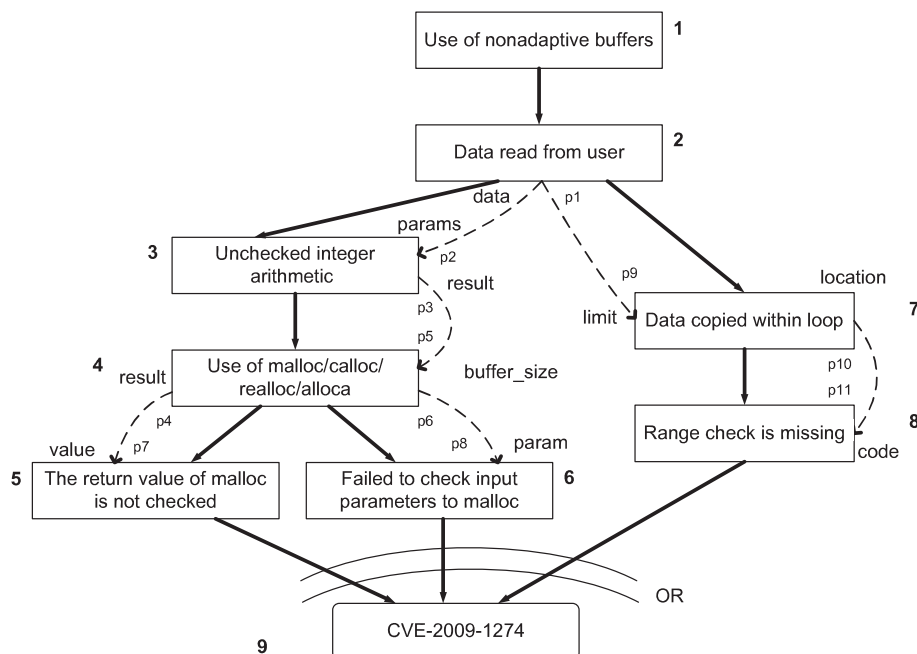


Fig. 7. SGM for a buffer overflow in *xine*.

Table 7
Condition templates for CVE-2009-1274.

Item	Vertex	Vertex	Vertex	Vertex	Vertex	Vertex
1. Vertex number	1	2	3	5	6	8
2. Previous vertex	No	1	2	4	4	7
3. Next vertex	2, 7	3	4	10	9	10
4. Search	Variable	Variable	Variable	Variable	Variable	Variable
5. Name	Buffer	Buffer_size	Buffer_size	Buffer	Buffer_size	Loop_counter
6. Type	Pointer	Integer	Integer	Pointer	Integer	Integer
7. Condition follows master action	No	No	No	Yes	No	No
8. Condition	Fixed	Result	Result	Unchecked	Unchecked	Unchecked
9. Condition element		User_input	Null	Null	Buffer_bounds	Counter_bounds

Table 8
Condition predicates for CVE-2009-1274.

Node	Predicate
1	Fixed (buffer)
2	Result (buffer_size, user_input)
3	Result (buffer_size, arithmetic)
5	Unchecked (buffer, null)
6	Unchecked (buffer_size, buffer_bounds)
8	Unchecked (loop_counter, counter_bounds)

the program. Note that the tool detects that bounds checking is done for a variable but, in the case of numeric variables, it does not check the values used in the program to perform the bounds checking.

Some conditions do not lend themselves to easy detection by the tool, such as the *nonAdaptiveBuffer* condition. Here, *TestInv-Code* asks the user to supply the information globally as input to the tool or specifically at points during the execution, where the condition is relevant.

Finally, *TestInv-Code* also maintains information on the VDCs that the user has selected. This means that for each VDC, the tool stores which clauses are satisfied and, if the proper set of clauses are satisfied (a master action has been detected and the proper pre- and postconditions for the VDC have been satisfied), the tool will report that the vulnerability modeled by the VDC has been found, indicating the address of the instruction in the executable or the line of source code (if debugging information is available), where this has happened.

7. Case study

Here we demonstrate the application of the vulnerability detection method to an open source application, *xine* (<http://www.xine-project.org>), which is written in C. We use an older version (xine-lib-1.1.15) of the application that contains the CVE-2009-1274 vulnerability described previously.

Starting from the SGM for this vulnerability, we have created the VDCs and the corresponding rules to be used for input to the *TestInv-Code* tool. The VDC was expressed using XML notation to facilitate their interpretation by the tool. We then executed *xine* under the control of *TestInv-Code*.

When the following *xine* code is executed we obtain the results shown in Fig. 8:

```
Code fragment from demux_qt.c
...
1907 trak->time_to_sample_table = calloc (
1908   trak->time_to_sample_count + 1,
        sizeof (time_to_sample_table_t));
1909 if (!trak->time_to_sample_table) {
1910   last_error = QT_NO_MEMORY;
1911   goto free_trak;
1912 }
1913
1914 /* load the time to sample table */
1915 for (j = 0; j < trak->time_to_sample_count; j++)
...

```

The field `trak->time_to_sample_table` is tainted since it is set from information taken from the external QuickTime file, which leads to the allocation on line 1907 possibly allocating the wrong amount of memory.

If we apply the same VDCs to other code, we should be able to find similar vulnerabilities, if they exist. To show this, we applied the same VDC on *ppmunbox*, an intentionally vulnerability-ridden

Table 9
Implementation of predicates in *TestInv-Code*.

C Instruction	Corresponding assembly code (what <i>TestInv-Code</i> analyzes)	Action performed
<code>x = malloc (y)</code>	<pre>mov eax,DWORD PTR [esp + 0x14] mov DWORD PTR [esp],eax call 8048388 <malloc@plt> mov DWORD PTR [esp + 0x10],eax</pre>	Memory location <code>esp + 0x10</code> is recognized as a dynamically allocated buffer and marked as unchecked
<code>strcpy (x,z)</code>	<pre>lea eax,[esp + 0x18] mov DWORD PTR [esp + 0x4],eax mov eax,DWORD PTR [esp + 0x10] mov DWORD PTR [esp],eax call 8048378 <strcpy@plt></pre>	If <code>esp + 0x10</code> is not marked as <i>checked</i> , then the allocation of <code>esp + 0x10</code> is <i>Unchecked</i> is satisfied for the allocation of <code>esp + 0x10</code> , and if the appropriate predicates are satisfied (this depends on the VDC) <i>TestInv-Code</i> may signal a vulnerability
<code>if (x == NULL)...</code>	<code>cmp DWORD PTR [esp + 0x10], 0x0</code>	The result of the <code>malloc</code> is being checked and the buffer at <code>esp + 0x10</code> will be marked as checked against <i>null</i>

program developed by Linköping University¹, and we detected the vulnerabilities shown in Fig. 9.

The relevant section of the *ppmunbox* source code is shown below. Here, a value read from a user-supplied file is used to calculate a buffer size. If the computation overflows, the buffer, allocated on line 92, will be too small for the data later read into it, resulting in an exploitable heap overflow.

```
Code fragment from ppmunbox.c
...
76:/* Read the dimensions */
77:if (fscanf (fp_in,"%d%d",& cols,& rows &
    maxval)<3){
78: printf ("unable to read dimensions from PPM
    file");
79: exit (1);
80:}
81:
82:/* Calculate some sizes */
83:pixBytes=(maxval > 255)? 6: 3;
84:rowBytes = pixBytes * cols;
85:rasterBytes = rowBytes*rows;
86:
87:/* Allocate the image */
88:img = malloc (sizeof (*img));
89:img->rows = rows;
90:img->cols = cols;
91:img->depth=(maxval > 255)?2:1;
92:p=(void*) malloc (rasterBytes);
93:img->raster = p;
94:
95:/* Read pixels into the buffer */
96:while (rows--) {
...

```

The use of *TestInv-Code* will cause the program being tested to run more slowly. If only a few VDCs (i.e. 10) are used, the slowdown is less than 30%, but if we have many VDCs (i.e. more than 100) applied to software that is data intensive (such as in the case of video decoders) *TestInv-Code* will slow down the execution significantly, to the point, where the program being tested could become unusable. To solve this problem *TestInv-Code* was modified so that the user can indicate specific functions to check in the program. This has two drawbacks. First, the user needs to know what functions need to be tested; and, second, all the input parameters for those functions need to be marked as tainted without the possibility of checking if they really are since the code that calls the function will not be analyzed by the tool. A better solution that is being explored by the authors is the possibility of checking only the first iteration of loops in the program, thus avoiding repeatedly checking code that is executed more than once. This makes the tool considerably faster and has been tested in the cases presented in this paper. It must be noted that at present, only applications that have ELF executables can be analyzed. In the future the tool will be able to analyze applications compiled using other formats (e.g. Windows Portable Executable (PE) format). Presently the tool can analyze programs that integrate different modules, plug-ins, pointers to functions, variable numbers of parameters but it does not allow for mixing different programming languages.

To illustrate the applicability and scalability of *TestInv-Code*, it has been applied to six different open source programs to

determine if known vulnerabilities can be detected using a single model. The following paragraphs describe the vulnerabilities and give a short explanation of the results obtained. The results are summarized in Table 10.

CVE-2009-1274. Integer overflow in the *qt_error_parse_trak_atom* function in *demuxers/demux_qt.c* in *xine-lib* 1.1.16.2 and earlier allows remote attackers to execute arbitrary code via a QuickTime movie file with a large count value in an STTS atom, which triggers a heap-based buffer overflow.

This vulnerability was detected as described earlier in this paper.

CVE-2004-0548. Multiple stack-based buffer overflows in the word-list-compress functionality in *compress.c* for *Aspell* allow local users to execute arbitrary code via a long entry in the word list that is not properly handled when using the (1) *c* compress option or (2) *d* decompress option.

Two vulnerabilities were detected by *TestInv-Code* using the VDC from the *xine* application:

```
do {
    *w++ = (char)(c);
} while (c = getc (in), c!= EOF && c > 32);
```

and

```
while ((c = getc (stdin)) > 32)
    cur[i++]=(char) c;
```

The loops depend on the data in a file and *w* points to a stack of size 256: *char sl[256]*. No check is made to avoid overflowing the buffer. Memory check tools like *Valgrind* could detect this error, but only if the input data used actually triggered the overflow. With *TestInv-Code* we are able to detect this type of error as long as the statements containing the error are executed (in this case, a loop that increments a pointer or an integer that is used to fill a table without any control); a failure does not have to occur.

CVE-2004-0557. Multiple buffer overflows in the *st_wavstar_tread* function in *wav.c* for *Sound eXchange (SoX)* 12.17.2 through 12.17.4 allow remote attackers to execute arbitrary code via certain WAV file header fields.

There is a vulnerability detected each time the function *st_reads* is called, where a loop depends on the length of data read from a file and *sc* points to a stack of size 256: *char text[256]*. No check is made to avoid overflowing the buffer. The only one made is with respect to the tainted value.

```
do {
    if (fread (& in, 1, 1, ft->fp) != 1) {
        *sc = 0;
        st_fail_errno (ft,errno,readerr);
        return (ST_EOF);
    }
    if (in == 0 || in == '\n') {
        break;
    }
    *sc = in; sc++;
} while (sc - c < len);
```

CVE-2004-0599. Multiple integer overflows in *png_read_png* in *pngread.c* or *png_handle_sPLT* functions in *pngutil.c* or progressive display image reading capability in *libpng* 1.2.5 and earlier

¹ *ppmunbox* processes portable pixmap (PPM) files, removing any single-color border around an image.


```
FOUND: use_of_tainted_not_bounded_value_to_alloc --> demuxers/demux_qt.c:1907
```

Fig. 8. Output from *TestInv-Code* run on *xine*.

```
FOUND: use_of_tainted_not_bounded_value_to_alloc --> pp/ppmunbox.c:92
```

Fig. 9. Output from *TestInv-Code* run on *ppmunbox*.

allow remote attackers to cause a denial of service (application crash) via a malformed PNG image.

TestInv-Code was able to find the vulnerabilities even here, where the program's reads, mallocs, etc. were redefined by custom functions, and the functions are called using pointers such as:

```
*(png_ptr->read_data_fn)(png_ptr, data, length);
```

Potential integer overflows such as:

```
info_ptr->row_pointers =
(png_bytepp) png_malloc (png_ptr,
info_ptr->height * sizeof (png_bytep));
```

were detected because `height` was marked as tainted. Note that some modifications to the tool were necessary so that it could work correctly when pointers to functions were used. This shows that the tool needs to be tested on different architectures to make certain that all cases work.

CVE-2008-0411. Stack-based buffer overflow in the `zseticc-space` function in `zicc.c` in Ghostscript 8.61 and earlier allows remote attackers to execute arbitrary code via a Postscript (.ps) file containing a long Range array in a `.seticcspace` operator.

As in the previous case, *TestInv-Code* was able to find the vulnerabilities even though reads, mallocs, etc. were redefined by custom functions, and an interpreted architecture is used.

CVE-2010-2067. Stack-based buffer overflow in the `TIFF-FetchSubjectDistance` function in `tif_dirread.c` in LibTIFF before 3.9.4 allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a long EXIF SubjectDistance field in a TIFF file.

This vulnerability was not detected by *TestInv-Code* because this type of vulnerability was not checked by the VDCs used so far. The authors have not yet determined how to detect this type of vulnerability using the currently available tool's techniques, since the program effectively does check for overflows in the data read (as shown in the code extract below), so it is not currently possible for the tool to detect the existence of potential problems.

```
/* Check for overflow. */
if (!dir->tdir_count || !w || cc/w != dir->tdir_count)
goto bad;
```

We conclude that *TestInv-Code* can effectively find known vulnerabilities but it needs to be improved to be able to manage software products that mix different programming languages, and, if we want to improve the tool's coverage, it needs to be extended to include other types of vulnerabilities, for instance, ones that involve using incorrect values when performing bound checks in the program. However, the tool allows detection of vulnerabilities using dynamic analysis of the code execution and the results obtained from the experiments show important improvement with respect to other

existing tools due to its flexibility (new VDCs can be designed), extensibility (new techniques can be introduced) and the precision of the VDCs (reducing the number of false positives).

8. Comparison of *TestInv-Code* to other tools

In this section we present a comparison of our method to others. No tool analyzes instruction sequences as our tool does. Our technique is different in that it is meant for detecting potential vulnerabilities during the execution mainly due to vulnerable program code. In this way it is possible to circumvent what Haugh and Bishop [29] call “the main problem with dynamic analysis”: the need for input data that causes overflows or errors to be able to detect them. With *TestInv-Code*, it is sufficient to execute the vulnerable code, but is not necessary to find inputs that trigger the vulnerability.

We compared *TestInv-Code* to TaintCheck [30], Daikon [31], splint [32] and Flawfinder [33]. Table 11 summarizes the differences between these tools, which can be considered complementary to our tool. Each targets vulnerabilities similar to those discussed in this paper. In all cases, the evaluation was conducted using only the model presented in this paper. This way we are able to verify that our tool detects this type of vulnerability, whereas the others do not (with the same input data). We are also able to estimate and compare the performance of the tools with respect to time spent executing them and the effort necessary to analyze the results (e.g. number of false positives that need to be discarded).

TaintCheck [30] is a taint analysis tool based on Valgrind [34], an instrumentation framework for building dynamic analysis tools. The authors of TaintCheck indicate a slowdown in execution of 37 times when analyzing *bzip2*. When compressing a 600 MB file under the control of TaintCheck, *bzip2* takes 9 min under the control of TaintCheck, compared to 10 s under the control of *TestInv-Code* (same file, same computer). This result is due to the loop iteration limiting parameter used in *TestInv-Code*, discussed in Section 7, which was set to 1000 iterations. Note that TaintCheck is not publicly available, so the evaluation of the tool is based on the authors' conclusions.

Daikon [31], also based on Valgrind, performs dynamic detection of likely program invariants. Vulnerabilities may be detected using the invariants that Daikon finds. Flawfinder [33] and splint [32] are popular static analysis tools that are capable of detecting program vulnerabilities similar to those detected using dynamic taint analysis.

Table 10

Summary of results running *TestInv-Code* with VDC for CVE-2009-1274.

Vulnerability	Software	Detected?
CVE-2009-1274	xine	Yes
Buffer overflow	ppmunbox	Yes
CVE-2004-0548	aspell	Yes (two)
CVE-2004-0557	SoX	Yes
CVE-2004-0559	libpng	Yes
CVE-2008-0411	Ghostscript	Yes
CVE-2010-2067	LibTIFF	No

Table 11

Qualitative comparison of the analysis tools that were tested.

	Daikon [31]	TestInv-Code	TaintCheck [30]	Splint [32]	Flawfinder [33]
Functionality	Dynamic detection of properties that are likely true at a particular point in a program (invariants)	Detects rules involving sequences of binary instructions being executed. Rules describe vulnerable programming	Dynamic taint analysis for automatic detection of overwrite attacks	Static checking of C programs for security vulnerabilities and programming mistakes. Uses code annotations	Pattern-matching-based static analysis for security vulnerabilities
Performance	Very slow	Fast if loop limitation is used	Author states slowdown of 1.5–40 times	Very fast	Very fast
Effectiveness	Analysis requires understanding of what the correct values for each variable is. Potential for many false positives	Adaption to each environment and compiler required. Indicated, where detected vulnerabilities are located. Easily extended to new vulnerabilities. Source code or special instrumentation of executable not required	No false positives. Detects and analyzes exploits, and improves automatic signature generation based on semantic analysis. Source code or special compilation not required	Requires annotations in source code to be effective. Detects many programmer errors apart from security issues. Typically produces a large number of results. Limited taint propagation capabilities [32]	Similar to splint

Table 12Detection results for *xine*.

Tool	Vulnerability detected	Others detected
TestInv-Code	Yes	0
Daikon	No	0
Splint	No	22 ^a
Flawfinder	No	41 ^b

^a Splint detected 22 *potential* vulnerabilities.^b Flawfinder detected 41 *potential* vulnerabilities.**Table 13**Detection results for *ppmunbox*.

Tool	Vulnerability detected	Others detected
TestInv-Code	Yes	0
Daikon	No	1058981 ^a
Splint	No	120 ^b
Flawfinder	No	41 ^c

^a Daikon detected the variables involved, but no values were given.^b Splint issued 120 warnings.^c Flawfinder detected 2 *potential* vulnerabilities.

We also considered evaluating *TestInv-Code* against Flayer [35] and Vulncheck [36], but Flayer does not perform sufficient analysis to detect vulnerabilities, and Vulncheck is no longer being maintained. We also omitted popular commercial tools such as Purify and Insure++ because of practical constraints.

Each of the tools was run against *xine* and *ppmunbox*, and the output was analyzed to determine if the tool had detected the vulnerability. For *xine*, the results are given in Table 12 and for *ppmunbox*, they are given in Table 13. In each case, *TestInv-Code* was able to detect the vulnerability, whereas the others were not, and *TestInv-Code* produced no false positives. The other tools detected additional vulnerabilities; these were not detected by *TestInv-Code* since it was run with a single model. Also note that automated active testing approaches, such as those used in SAGE [37] or EXE [38] would almost certainly have fared better in these tests, since they are designed to find good inputs for security testing.

Note that dynamic analysis tools like the ones tested here are only able to detect problems if the input data triggers the problem. In this case, the input data (the same in each case) executed the vulnerable code but did not trigger the vulnerability. Since *TestInv-Code* is based on detecting potentially vulnerable execution patterns (as defined by a security goal model), it is able to detect the problem even if it is not triggered by the input data.

9. Related work

Different techniques have been proposed to perform dynamic detection of vulnerabilities. Fuzz testing is an approach that has been proposed to improve the security and reliability of system implementations [39]. Fuzz testing consists of changing the inputs, using random inputs or mutation techniques, in order to detect unwanted behavior, such as crashing or confidentiality violation. Penetration testing is another technique that consists of executing a predefined test scenario with the objective of detecting design vulnerabilities or implementation vulnerabilities [40].

Fault injection is a similar technique that injects different types of faults in order to test the behavior of the system [41]. After injecting one or more faults, the system behavior is observed. The failure to tolerate faults is an indicator of a potential security flaw in the system. These techniques have been applied in industry and shown to be useful. However, most of the current detection techniques based on these approaches are ad hoc and require a previous knowledge of the target systems or existing exploits.

In the dynamic taint approach of Chess and West [42], tainted data are monitored during the execution of the program to determine its proper validation before entering sensitive functions. It enables the discovery of possible input validation problems which are reported as vulnerabilities. The sanitization technique to detect vulnerabilities due to the use of user supplied data is based on the implementation of new functions or custom routines. The main idea is to validate or sanitize any input from the users before using it inside a program. Balzarotti et al. [43] present an approach using static and dynamic analysis to identify faulty sanitization processes (sanitization that an attacker can bypass) in web applications.

The approach proposed in this paper is close to that of dynamic taint analysis. However, it is original because it covers all the steps of vulnerability detection, from the modeling of vulnerabilities using SGMs, which are close to the users' requirements, and VDCs, which provide a formal description facilitating their automated detection on the traces of the program's execution using the *TestInv-Code* tool. This methodology also allows detecting vulnerability causes other than tainted data, such as lack of bound control of variables, bad memory allocation and use, and so forth.

An alternate approach to software security is to prove, in some way, the absence of vulnerabilities. Such methods are often based on formally specifying system requirements or design, including security aspects, then either generating the system or verifying that the system (or a model of the system) conforms to the specification. Notable examples of this approach include Event-B [44] (in which systems are generated from Event-B specifications);

Secure Tropos [45], which extends the Tropos [46] agent-oriented software development method with security-related models and activities; UMLSec [47]; various approaches for specifying and analyzing security protocols, access control policies and other system properties [48–53].

Model checking techniques have also been revisited for vulnerability detection. Hadjidi et al. present a security verification framework that uses a conventional push down system model checker for reachability properties to verify software security properties [54]. Wang et al. have developed a constraint analysis combined with model checking in order to detect buffer overflow vulnerabilities [55]. The memory size of buffer-related variables is traced and the code instrumented with constraint assertions before the potential vulnerable points. The vulnerability is then detected with the reachability of the assertion using model checking. All model checking is based on the design of a model of the system, which can be complex and subject to the combinatorial explosion of the number of states.

Our testing approach differs from most formal methods in several ways. The testing techniques proposed by EXE [38] and SAGE [37] are based on active testing techniques that consist in stimulating an implementation using selected inputs. These tools are designed to find good inputs for security testing. Our approach, on the other hand, is based on passive testing techniques. It does not explicitly stimulate the implementation but consists mainly of observing the implementation executing with normal inputs, and collecting traces. Most importantly, our approach is independent of the specification or design of the system under test, since it primarily targets typical implementation errors; it is not aimed at requirements or design issues. Furthermore, our approach has a very modest impact on the development process, and can easily be used in any traditional process; many formal methods have a far greater impact on the process.

Following the testing approach defined in this paper, the coverage is defined with respect to VDCs that are evaluated on a corpus of traces. Formal approaches to software security can provide more comprehensive coverage of security than our approach does, but that comes at a cost. Furthermore, formal methods that do not guarantee the security of the implementation must be combined with some assessment technique, such as passive testing.

SGMs are reminiscent of fault trees [56,57], and attack trees [15], but the inclusion of information edges and the ability to model subgoals make SGMs more expressive than either of the earlier languages. Furthermore, attack trees lack a formal underpinning (although attempts have been made to correct this [58]). Other languages for modeling goals exist, such as goal diagrams in Tropos [46], but the complexity and actor-centric approach of these make them inappropriate for SGM applications. Other threat modeling languages, such as data flow diagrams [59] and misuse case diagrams [60–62] and extended use case diagrams [63] are appropriate to address security in requirements and design, but are not applicable to the kind of testing discussed in this paper.

10. Conclusions and future work

Security has become a critical part of nearly every software project, and the use of automated testing tools is recommended by best practices and guidelines. Our interest lies in extending the concept of security goal models (SGMs) so we can detect the vulnerabilities they model using automated testing. In addition to the work presented here, we are exploring the use of SGMs as a basis for static analysis for vulnerability detection.

In this paper we have presented a formalization of SGMs and their causes called vulnerability detection conditions (VDCs) that can be used in automated testing to detect the presence of

vulnerabilities. We have also shown how this is applied in a passive testing tool, *TestInv-Code*, which analyzes execution traces to determine if they show evidence of a vulnerability or not. VDCs can be very precise, which we believe makes it possible to detect vulnerabilities with a low rate of false positives.

Since the vulnerability models are separate from the tool, it is possible for anyone, not just the tool vendor, to keep them up-to-date and to add new models. It also becomes possible for the tool user to add, e.g., product-specific vulnerabilities and use the tool to detect them. This is very different from the normal state of affairs, where users have no choice but to rely on the tool vendor to provide timely updates. As stated before, currently the tool allows users to define new VDCs. It also allows implementing new predicates that are linked to the tool using the tool's plugin architecture, but this is not yet easily done by a user since it requires knowledge of the tool's implementation.

The work presented in this paper was part of the SHIELDS EU project [10], in which we have developed a shared security repository through which security experts can share their knowledge with developers by using security models. Models in the SHIELDS repository are available to a variety of development tools; *TestInv-Code* is one such tool. Further funding was provided through the Software Intensive Systems program of the Swedish Foundation for Strategic Research.

Looking to the future, we plan on applying the methods presented here to various kinds of vulnerabilities in order to identify which predicates are required, and whether the formalism needs to be extended in some way. We will also further develop *TestInv-Code* to handle more complex applications and allow users to easily define new predicates.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under Grant agreement no. 215995. Further funding was provided through the Software Intensive Systems program of the Swedish Foundation for Strategic Research.

References

- [1] CERT Coordination Center, CERT/CC Statistics. <<http://www.cert.org/stats/>> (accessed April 2011).
- [2] E. Bayse, A.R. Cavalli, M. Núñez, F. Zaïdi, A passive testing approach based on invariants: application to the WAP, Computer Networks and ISDN Systems 48 (2) (2005) 247–266, <http://dx.doi.org/10.1016/j.comnet.2004.09.009>.
- [3] C. Kuang, Q. Miao, H. Chen, Analysis of software vulnerability, in: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy (ISP'06), World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2006, pp. 218–223.
- [4] S. Redwine, N. Davis, Processes to produce secure software, Task Force on Security Across the Software Development Lifecycle, 2004, Appendix A, 2004.
- [5] Coverity, Prevent. <<http://www.coverity.com/>> (accessed April 2011).
- [6] Fortify Software, Fortify 360 source code analyzer. <<http://www.fortifysoftware.com/products>> (accessed April 2011).
- [7] Klocwork, K7. <<http://www.klocwork.com/>> (accessed April 2011).
- [8] D. Byers, N. Shahmehri, Unified modeling of attacks, vulnerabilities and security activities, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS'10), IEEE Computer Society, Washington, DC, USA, 2010, pp. 36–42, <http://dx.doi.org/10.1145/1809100.1809106>.
- [9] D. Byers, S. Ardi, N. Shahmehri, C. Duma, Modeling software vulnerabilities with vulnerability cause graphs, in: Proceedings of the International Conference on Software Maintenance (ICSM06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 411–422. doi:10.1109/ICSM.2006.40.
- [10] SHIELDS: Detecting known vulnerabilities from within design and development tools, a project within the EU's seventh framework programme. <<http://www.shields-project.eu/>>.
- [11] Final report on inspection methods and prototype vulnerability recognition tools, SHIELDS Deliverable D4.3, April 2010.
- [12] D. Byers, N. Shahmehri, Design of a process for software security, in: Proceedings of the Second International Conference on Availability, Reliability and Security (ARES'07), IEEE Computer Society, Washington, DC, USA, 2007, pp. 301–309. doi:10.1109/ARES.2007.67.

- [13] D. Byers, N. Shahmehri, A cause-based approach to preventing software vulnerabilities, in: S.T. Stefan Jakoubi, E.R. Weippl (Eds.), *Proceedings of the Third International Conference on Availability, Reliability and Security (ARES'08)*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 276–283. doi:10.1109/ARES.2008.12.
- [14] H. Peine, M. Jawurek, S. Mandel, Security goal indicator trees: A model of software features that supports efficient security inspection, in: *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 9–18. doi:10.1109/HASE.2008.57.
- [15] B. Schneier, *Attack Trees*, Dr. Dobbs Journal, December 1999.
- [16] D. Harel, B. Rumpe, Meaningful modeling: what's the semantics of "semantics"? , IEEE Computer 37 (10) (2004) 64–72, <http://dx.doi.org/10.1109/MC.2004.172>.
- [17] Final modelling methods, formalisms and prototype modelling tools, SHIELDS Deliverable D2.3, April 2010.
- [18] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, T. Oinn, Taverna: a tool for building and running workflows of services, *Nucleic Acids Research* 34 (Web Server Issue) (2006) 729–732.
- [19] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about datalog (and never dared to ask), *IEEE Transactions on Knowledge and Data Engineering* 1 (1) (1989) 146–166, <http://dx.doi.org/10.1109/TKDE.1989.924410>.
- [20] A.R. Cavalli, D. Vieira, An enhanced passive testing approach for network protocols, in: *Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICN/ICSMCL'06)*, 2006, pp. 169–169. doi:10.1109/ICN/ICSMCL.2006.50.
- [21] B. Alcalde, A.R. Cavalli, D. Chen, D. Khuu, D. Lee, Network protocol system passive testing for fault management: A backward checking approach, in: *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2004)*, 2004, pp. 150–166. doi:10.1007/b100576.
- [22] D. Lee, A. Netravali, K. Sabnani, B. Sugla, A. John, Passive testing and applications to network management, in: *Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, IEEE Computer Society, Washington, DC, USA, 1997, p. 113.
- [23] R.E. Miller, K.A. Arisha, Fault identification in networks by passive testing, in: *Proceedings of the 34th Annual Simulation Symposium (SIMSYM'01)*, IEEE Computer Society, 2001, pp. 277–284, <http://dx.doi.org/10.1109/SIMSYM.2001.922142>.
- [24] A.R. Cavalli, C. Gervy, S. Prokopenko, New approaches for passive testing using an extended finite state machine specification, *Information and Software Technology* 45 (12) (2003) 837–852, [http://dx.doi.org/10.1016/S0950-584\(03\)00063-6](http://dx.doi.org/10.1016/S0950-584(03)00063-6).
- [25] W. Mallouli, F. Bessayah, A.R. Cavalli, A. Benameur, Security rules specification and analysis based on passive testing, in: *The IEEE Global Communications Conference (GLOBECOM 2008)*, 2008, pp. 1–6, doi:10.1109/GLOBECOM.2008.ECP.400.
- [26] M. Tabourier, A.R. Cavalli, Passive testing and application to the GSM-MAP protocol, *Information and Software Technology* 41 (11–12) (1999) 813–821, [http://dx.doi.org/10.1016/S0950-584\(99\)00039-7](http://dx.doi.org/10.1016/S0950-584(99)00039-7).
- [27] Final repository specification design and prototype, SHIELDS Deliverable D3.3, April 2010.
- [28] A.R. Cavalli, E. Montes de Oca, W. Mallouli, M. Lallali, Two complementary tools for the formal testing of distributed systems with time constraints, in: *The 12th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'08)*, IEEE Computer Society, Vancouver, Canada, 2008, pp. 315–318. doi:10.1109/DS-RT.2008.43.
- [29] E. Haugh, M. Bishop, Testing C programs for buffer overflow vulnerabilities, in: *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS'03)*, 2003, pp. 123–130.
- [30] J. Newsome, D. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: *Proceedings of the 2005 Network and Distributed System Symposium (NDSS'05)*, 2005.
- [31] M.D. Ernst, J.H. Perkins, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming* 69 (1–3) (2007) 35–45, <http://dx.doi.org/10.1016/j.scico.2007.01.015>.
- [32] D. o. C.S. University of Virginia, Splint: Annotation-assisted lightweight static checking. <<http://www.splint.org/>> (accessed September 2011).
- [33] D.A. Wheeler, Flawfinder. <<http://www.dwheeler.com/flawfinder/>> (accessed September 2011).
- [34] Valgrind. <<http://valgrind.org/>> (accessed September 2011).
- [35] W. Drewry, T. Ormandy, Flayer: Exposing application internals, in: *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT'07)*, 2007.
- [36] A.I. Sotirov, Automatic Vulnerability Detection Using Static Source Code Analysis, Master's thesis, The University of Alabama, 2005.
- [37] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, M.Y. Levin, Automating software testing using program analysis, *IEEE Software* 25 (2008) 30–37, <http://dx.doi.org/10.1109/MS.2008.109>.
- [38] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, EXE: Automatically generating inputs of death, in: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, 2006. doi:10.1145/1455518.1455522.
- [39] M. Howard, Inside the windows security push, in: *IEEE Symposium on Security and Privacy*, 2003, pp. 57–61. doi:10.1109/MSECP.2003.1176996.
- [40] H. Thompson, Application penetration testing, *IEEE Security and Privacy* 3 (1) (2005) 66–69, <http://dx.doi.org/10.1109/MSP.2005.3>.
- [41] W. Du, A. Mathur, Vulnerability testing of software system using fault injection, in: *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, Workshop on Dependability Versus Malicious Faults, 2000.
- [42] B. Chess, J. West, Dynamic taint propagation: finding vulnerabilities without attacking, *Information Security Technical Report* 13 (1) (2008) 33–39, <http://dx.doi.org/10.1016/j.istr.2008.02.003>.
- [43] D. Balzarotti, M. Cova, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: composing static and dynamic analysis to validate sanitization in web applications, in: *IEEE Symposium on Security and Privacy*, 2008, pp. 387–401. doi:10.1109/SP.2008.22.
- [44] J.-R. Abrial, M. Butler, S. Hallerstede, L. Voisin, An open extensible tool environment for event-B, in: *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM'06)*, Lecture Notes in Computer Science, vol. 4260/2006, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 588–605. doi:10.1007/11901433_32.
- [45] P. Giorgini, H. Mouratidis, N. Zannone, Modelling security and trust with secure tropos, in: *Integrating Security and Software Engineering: Advances and Future Visions*, Idea Group Publishing, 2007, pp. 160–189.
- [46] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos, Tropos: an agent-oriented software development methodology, *Autonomous Agents and Multi-Agent Systems* 8 (3) (2004) 203–236, <http://dx.doi.org/10.1023/B:ACNT.0000018806.20944.ef>.
- [47] J. Jürjens, *Secure Systems Development with UML*, Springer, Berlin, Heidelberg, 2005.
- [48] D. Basin, S. Mödersheim, L. Viganò, OFMC: a symbolic model checker for security protocols, *International Journal of Information Security* 2005 (4) (2005) 181–208, <http://dx.doi.org/10.1007/s10207-004-0055-7>.
- [49] A. Armando, L. Compagna, SAT-based model-checking for security protocols analysis, *International Journal of Information Security* 7 (1) (2008) 3–32, <http://dx.doi.org/10.1007/s10207-007-0041-y>.
- [50] T. Lodderstedt, D.A. Basin, J. Doser, SecureUML: A UML-based modeling language for model-driven security, in: *Proceedings of the 5th International Conference on The Unified Modeling Language (UML'02)*, Springer-Verlag, London, UK, 2002, pp. 426–441. doi:10.1007/3-540-45800-X_33.
- [51] L. Viganò, Automated security protocol analysis with the AVISPA tool, *Electronic Notes in Theoretical Computer Science* 155 (2006) 61–86. *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*. doi:10.1016/j.entcs.2005.11.052.
- [52] K. Fisler, S. Krishnamurthi, L.A. Meyerovich, M.C. Tschantz, Verification and change-impact analysis of access-control policies, in: *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, ACM, New York, NY, USA, 2005, pp. 196–205. doi:10.1145/1062455.1062502.
- [53] N. Zhang, M. Ryan, D.P. Guelev, Evaluating access control policies through model checking, in: *Proceedings of the 8th International Conference on Information Security (ISC 2005)*, Lecture Notes in Computer Science, vol. 3650, Springer, Berlin, Heidelberg, 2005, pp. 446–460. doi:10.1007/11556992_32.
- [54] R. Hadjidi, X. Yang, S. Tlili, M. Debbabi, Model-checking for software vulnerabilities detection with multi-language support, in: *Sixth Annual Conference on Privacy, Security and Trust*, 2008, pp. 133–142. doi:10.1109/PST.2008.21.
- [55] L. Wang, Q. Zhang, P. Zhao, Automated detection of code vulnerabilities based on program analysis and model checking, in: *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 165–173. doi:10.1109/SCAM.2008.24.
- [56] C.A. Ericson II, Fault tree analysis – a history, in: *Proceedings of the 17th International System Safety Conference*, 1999.
- [57] D. Haas, Advanced concepts in fault tree analysis, in: *System Safety Symposium*, 1965.
- [58] S. Mauw, M. Oostdijk, Foundations of attack trees, in: *Information Security and Cryptology (ICISC 2005)*, Lecture Notes in Computer Science, vol. 3935/2006, Springer-Verlag, 2006, pp. 186–198. doi:10.1007/11734727_17.
- [59] F. Swiderski, W. Snyder, *Threat Modeling*, Microsoft Press, 2004.
- [60] J. McDermott, C. Fox, Using abuse case models for security requirements analysis, in: *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC'99)*, IEEE Computer Society, Washington, DC, USA, 1999, p. 55. doi:10.1109/CSAC.1999.816013.
- [61] L. Rostad, An extended misuse case notation: Including vulnerabilities and the insider threat, in: *The 12th Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'06)*, Essener Informatik Beiträge, Essen, Germany, 2006, pp. 33–43.
- [62] G. Sindre, A.L. Opdahl, Eliciting security requirements with misuse cases, *Requirements Engineering* 10 (1) (2005) 34–44, <http://dx.doi.org/10.1007/s00766-004-0194-4>.
- [63] G. Popp, J. Jürjens, G. Wimmel, R. Breu, Security-critical system development with extended use cases, in: *Tenth Asia-Pacific Software Engineering Conference*, 2003, pp. 478–487. doi:10.1109/APSE.2003.1254403.