available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Extending the enforcement power of truncation monitors using static analysis

Hugues Chabot, Raphaël Khoury, Nadia Tawbi*

Département d'informatique et de génie logiciel, Université Laval, 1065, av. de la Médecine, Québec City, Québec, Canada G1V 0A6

ARTICLE INFO

Article history:
Received 29 June 2010
Received in revised form
23 October 2010
Accepted 21 November 2010

Keywords:
Computer security
Dynamic analysis
Monitoring
Software safety

ABSTRACT

Runtime monitors are a widely used approach to enforcing security policies. Truncation monitors are based on the idea of truncating an execution before a violation occurs. Thus, the range of security policies they can enforce is limited to safety properties. The use of an a priori static analysis of the target program is a possible way of extending the range of monitorable properties. This paper presents an approach to producing an in-lined truncation monitor, which draws upon the above intuition. Based on an a priori knowledge of the program behavior, this approach allows, in some cases, to enforce more than safety properties and is more powerful than a classical truncation mechanism. We provide and prove a theorem stating that a truncation enforcement mechanism considering only the set of possible executions of a specific program is strictly more powerful than a mechanism considering all the executions over an alphabet of actions.

1. Introduction

Execution monitoring is an approach to enforcing security policies that seeks to allow an untrusted code to run safely by observing its execution and reacting if necessary to prevent a potential violation of a user-supplied security policy. This method has many promising applications, particularly with respect to the safe use of mobile code.

Academic research on monitoring has generally focused on two questions. The first relates to the set of policies that can be enforced by monitors and the conditions under which this set could be extended. The second question deals with the way to in-line a monitor in an untrusted or potentially malicious program in order to produce a new instrumented program that provably respects the desired security policy.

While studies on security policy enforcement mechanisms show that an a priori knowledge of the target program's behavior would increase the power of these mechanisms (Hamlen et al., 2006; Bauer et al., 2002), no further investigations have been pursued in order to take full advantage of this idea in the context of runtime monitoring. As a result, implementations of truncation based monitoring frameworks remain limited in the set of policies that they can enforce to the set of safety properties.

This paper, presents an approach to generate a safe instrumented program, from a security policy and an untrusted program in which the monitor draws on an a priori knowledge of the program's possible behavior. This allows the monitor to sometimes enforce non-safety properties, which were beyond the scope of previous approaches.

This approach draws on advances in discrete events system control by Ramadge and Wonham (1989) and on related subsequent research by Langar and Mejri (2005) and consists in combining two models via the automata product operator: a model representing the system's behavior and another one representing the property to be enforced. In this

* *Corresponding author*. Tel.: +1 418 653 0177.
E-mail addresses: hugues.chabot.1@ulaval.ca (H. Chabot), raphael.khoury.1@ulaval.ca (R. Khoury), nadia.tawbi@ift.ulaval.ca (N. Tawbi).
0167-4048/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2010.11.004

approach, the system's behavior is modeled by an LTS and the property to be enforced is stated as a Rabin automaton, a model which can recognize the same class of languages as non-deterministic Büchi automata (Perrin and Pin, 2004). A major advantage of this representation over alternatives such as the Büchi automaton is its determinism, which simplifies the method and the associated proofs.

The algorithm either returns an instrumented program that provably respects the input security policy, otherwise it terminates with an error message. While the latter case sometimes happens, it is important to stress that this will never occur if the desired property is a safety property which can be enforced using existing approaches. Indeed, any safety property is still enforced by truncation as is presently the case in preceding works. When enforcing a non-safety property, the approach relies both on static program transformations and runtime truncation. The approach presented in this paper is thus strictly more expressive. Furthermore, this increase in the set of enforceable property is achieved without relying on the transformative capacities of the edit automaton, and without imposing a runtime overhead to the execution.

The rest of this paper is organized as follows. Section 2 presents a review of related work. Section 3 defines some concepts that are used throughout the paper. The elaborated method is presented in Section 4. Section 5 discusses the theoretical underpinnings of the method. Some concluding remarks are finally drawn in Section 6 together with an outline of possible future work.

2. Related work

Schneider, in his seminal work (Schneider, 2000), was the first to investigate the question of which security policies could be enforced by monitors. He focused on specific classes of monitors, which observe the execution of a target program with no knowledge of its possible future behavior and with no ability to affect it, except by aborting the execution. Under these conditions, he found that a monitor could enforce precisely those security policies that are identified in the literature as safety properties, and are informally characterized by prohibiting a certain bad thing from occurring in a given execution. These properties can be modeled by a security automaton and their representation has formed the basis of several practical as well as theoretical monitoring frameworks.

Schneider's study also suggested that the set of properties enforceable by monitors could be extended under certain conditions. Building on this insight, Bauer et al. (2002) and Ligatti et al. (2005a) examined the way the set of policies enforceable by monitors would be extended if the monitor had some knowledge of its target's possible behavior or if its ability to alter that behavior were increased. The authors modified the above definition of a monitor along three axes, namely (1) the means on which the monitor relies in order to respond to a possible violation of the security policy; (2) whether the monitor has access to information about the program's possible behavior; (3) and how strictly the monitor is required to enforce the security policy. Consequently, they were able to provide a rich taxonomy of classes of security policies, associated with the appropriate model needed to enforce them. Several of these models are strictly more powerful than the security automata developed by Schneider and are used in practice.

Evolving along this line of inquiry, Ligatti et al. (2005b) gave a more precise definition of the set of properties enforceable by the most powerful monitors, while Fong (2004) and Talhi et al. (2008) expounded on the capabilities of monitors operating under memory constraints. Hamlen et al. (2006), on the other hand showed that in-lined monitors (whose operation is injected into the target program's code, rather than working in parallel) can also enforce more properties than those modeled by a security automaton. In Bauer et al. (2006), a method is given to enforce both safety and co-safety properties by monitoring. Alternative definitions of enforcement were given in Bielova et al. (2009), Khoury and Tawbi (in press) and Ligatti and Reddy (2010).

The first practical application using this framework was developed by Erlingsson and Schneider (2000). In that project, a security automaton is merged into object code, and static analysis is used to reduce the runtime overhead incurred by the policy enforcement. Similar approaches, working on source code, were developed by Colcombet and Fradet (2000), by Langar and Mejri (2005) and by Kim (2001), Kim et al. (2004), Lee et al. (1999), Sokolsky et al. (1999). All these methods are limited to enforcing safety properties, which must be included either as a security automaton, or stated in a custom logic developed for this application. The first two focus on optimizing the instrumentation introduced in the code.

3. Preliminaries

Before moving on, let us briefly start with some preliminary definitions.

The desired security property is stated as a Rabin automaton. The Rabin automaton is a finite state automaton which can recognize infinite-length sequences. The input sequence is recognized if it satisfies an acceptance condition, given by a set of pairs of sets of states, which restricts the states which an accepted sequence can visit infinitely often. When the Rabin automaton models a security property, the accepted sequences correspond to the sequences satisfying the property. A Rabin automaton $\mathcal{R}$, over alphabet $\mathcal{A}$ is a tuple (Q,q_0,δ,C) such that

- $\mathcal{A}$ is a finite or countably infinite set of symbols;
- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $\delta \subseteq Q \times A \times Q$ is a transition function;
- $C = \{(L_j,U_j)|j \in J\}$ is the acceptance set. It is a set of couples (L_j,U_j) where $L_j \subseteq Q$ and $U_j \subseteq Q$ for all $j \in J$ and $J \subseteq \mathbb{N}$.

Let $\mathcal{R}$ stand for a Rabin automaton defined over alphabet $\mathcal{A}$. A subset $Q' \subseteq Q$ is *admissible* if and only if there exists a $j \in J$ such that $Q' \cap L_j = \emptyset$; and $Q' \cap U_j \neq \emptyset$.

For the sake of simplicity, the elements defining an automaton or a model are referred to using the following formalism: the set of states Q of automaton $\mathcal{R}$ is referred to as $\mathcal{R}\cdot Q$ or simply Q when $\mathcal{R}$ is clear from the context.

A *path* π, is a finite (respectively infinite) sequence of states $\langle q_1,q_2,\ldots,q_n\rangle$ (respectively $\langle q_1,q_2,\ldots\rangle$) such that there exists a finite (respectively infinite) sequence of symbols $a_1,a_2,\ldots,a_n$