Information and Software Technology 73 (2016) 19–36

Contents lists available at ScienceDirect

Information and Software Technology

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

A low-overhead, value-tracking approach to information flow security[☆]

Kostyantyn Vorobyov[a,*], Padmanabhan Krishnan[b], Phil Stocks[a]

[a] Centre for Software Assurance, Bond University, Gold Coast, Australia
[b] Oracle Labs, Brisbane, Australia

A R T I C L E I N F O

Article history:
Received 12 April 2014
Revised 9 December 2015
Accepted 15 December 2015
Available online 21 January 2016

Keywords:
Information leakage
Program instrumentation
Monitoring

A B S T R A C T

Context: Sensitive information such as passwords often leaks inadvertently because of implementation defects.

Objective: Our objective is to use dynamic techniques to prevent information leakage before it occurs. We also aim to develop techniques that incur low overheads, and are safe in the presence of aliasing.

Method: We use a dynamic approach to track secret values and safe locations. We assume that programs have annotations which identify values and locations that need to be protected against disclosure. We instrument a program with statements that record relevant values and locations and assertions to relevant assignments to determine if they leak information. At run-time the values being assigned to unsafe locations are analysed. If a particular assignment leads to information leakage an assertion violation is triggered. We evaluate our approach by experimentation which uses our prototype implementation for C programs to analyse security-oriented UNIX utilities and programs chosen from the SPEC CPU datasets.

Results: Our experiments show that the overhead to detect problems such as password disclosure in real software does not exceed 1%. The overheads associated with detection of CWE security vulnerabilities in real applications are still acceptable; however, tracking a large number of values incurs higher overheads (over 10 times in certain cases).

Conclusion: Our dynamic approach to detecting information leaks can be used in various contexts. For a program that tracks only a limited number of values the overhead is marginal. Thus, our instrumentation can be used in release versions. However, if an application has a large number of secret values, our technique is useful in a testing phase. The overheads in this case are too high for a real use, but still within an acceptable range to be used for detection of potential leaks during testing.

1. Introduction

The problem addressed by information leakage detection is to ensure that data (i.e., a set of values in a program run) identified as secret are not exposed externally, for example through a publicly visible variable, or direct output by a print function. If the secret values in a run of a program are known, the program can be checked for information leakage by calculating dependencies between secret and publicly available data.

In imperative languages values are accessed by evaluating variables, which often reference actual values rather than storing them. While it is possible to over-approximate the set of values bound to variables using pure static methods, it is not possible to compute this exactly in general as the problem is undecidable [2]. Imprecise and scalable approximations, on the other hand, often lead to false alarms. Since at run-time values are known, aliasing can be solved by dynamically tracking assignments of values to variables during a program run. Therefore, a dynamic approach, such as information flow [3–5] or taint analysis [6,7] is appropriate for detection of information leakage. This is because such a technique captures every assignment and evaluates values for a particular program run. Tracking every assignment, however, often results in high overhead [5,8,9].

The key research question we address is the design and implementation of run-time monitors that are able to detect and prevent hitherto unknown violations of information flow in real programs. This technique prevents disclosure of confidential information at run-time by aborting the execution of a program run that leaks secret data. We also address the question of reducing the high overheads that are often associated with tracking the flow of sensitive data at execution time. As noted by others [10–12], run-time mon-

☆ An initial version of this paper was presented at SEFM'12 [1].
* Corresponding author. Tel.: +61 401787874, +33 768139944.
E-mail addresses: kvorobyo@bond.edu.au (K. Vorobyov), paddy.krishnan@oracle.com (P. Krishnan), pstocks@bond.edu.au (P. Stocks).

http://dx.doi.org/10.1016/j.infsof.2015.12.006
0950-5849/© 2015 Elsevier B.V. All rights reserved.

itoring of programs that incur heavy overhead often restricts its usage to only a subset of realistic programs. In the absence of verified programs, run-time monitoring with a low overheads permits the detection of unknown information flow violations in realistic programs.

Our technique analyses program values and has the ability to identify whether a disclosed value represents an information leak with respect to the values considered secret at run-time. This is different to an information flow or a taint analysis that typically analyses a program with respect to its variables and tracks security labels or propagates taint marks. Tracking only a handful of values whose disclosure constitutes information leakage reduces the overheads associated with discovering leakage. The scope of our analysis is therefore to detect leakage of values in their entirety. Detection of values that leak via parts (e.g., bit by bit) is out of scope of this paper.

We assume that the statements that generate secret data are identified, for example using manual annotations to the program. We view such annotations as an input to our technique, thus the question of identification of secret data is out of the scope of this paper. The annotations are used to instrument the program with statements that record secret values that should not leak. Further, for any potentially leaking assignment we inject an assertion that fails if a concrete value used in the assignment belongs to the tracked set of secret values. During its execution, the instrumented program captures the secret values and uses assertions to verify the safety of potentially leaking assignments with respect to the secret values received by the program for a particular run.

Our approach has been applied to an unsafe memory model. It detects leakage in programs written in memory-unsafe languages, e.g., C, where precise information leakage analysis is a challenge due to such features as pointer arithmetic, weak type system or dynamic memory allocation. The question of precise and scalable information leakage analysis in memory-unsafe environments is not addressed in full by the existing techniques that either use safe models [13,14], or restrict the language features to a manageable subset [15]. Note that our approach does not prevent memory corruption errors such as buffer overflows, it only prevents secret values from leaking assuming that a program under analysis is memory safe.

Our approach is supported by a prototype implementation for C programs that is used to conduct experiments. The results of our initial experiments show that our approach can be used to address practical problems, such as preventing leakage of passwords. Monitoring the safety of password flow in a number of security-oriented UNIX utilities indicates that this dynamic analysis of secret values results in an extremely low overhead of 1% and finds information leakage in real security software. Further experimentation demonstrates that our technique is a good fit for analysing programs for security vulnerabilities related to information leakage stressed by security-oriented communities, such as the Community Developed Dictionary of Software Weakness Types (CWE) [16]. We also report on our experiments using a number of computationally expensive programs from the SPEC datasets. This addresses issues related to information leaks via the de-allocated but not cleared out memory, improper handling of sensitive data (e.g., plain-text storage or hard-coding of passwords), exposure of sensitive information through standard output channels and information leaks via temporary files and file handles. The results show that our approach handles complex programs, such as `gcc`, while still yielding acceptable overheads. Finally, we use the same properties to analyse popular security-oriented software such as `openssh` and `ccrypt` and show that the overheads incurred by our approach remain low.

The rest of the paper is organised as follows. Section 2 presents our technique at an abstract level and Section 3 shows how to apply it to C programs. Section 4 presents the results of the experimentation with the prototype implementation of our technique. Finally, Section 5 reviews related work and Section 6 offers our conclusions.

2. Value tracking

Our approach assumes that secret values are specified by annotations. That is, program locations of assignments that transfer secret values to program variables are marked. An input program $\boldsymbol{P}$ is instrumented (by a series of source-to-source transformations) with statements that track secret values and safe locations, and assertions that check the safety of assignments with respect to the tracked values. This generates a monitored program $\boldsymbol{P'}$. A run of $\boldsymbol{P'}$ observes the execution of the original program $\boldsymbol{P}$ by detecting information leakage by assignment of secret values to unsafe locations. A program run that has no detected assertion failures does not leak the secret values captured at run-time by the annotated assignments.

In this section we present our approach for an abstract imperative language. We first describe the details of the imperative language used to specify transformations, the abstract memory model, and the operational semantics of the language. We then informally describe our approach and present the set of compositional transformation rules used to derive a monitored program that prevents leakage of secret values at run-time. Further, we give the semantics of instrumented commands and describe behaviours of monitored programs. Finally, we present a proof that our monitor does not interfere with the memory state of the original program and prevents all information leaks that occur due to assignment of secret data to unsafe locations.

2.1. Syntax

Fig. 1 shows the syntax of the abstract imperative language we use to describe our approach. Variables v (given by the set *Var*) are partitioned into variables that hold primitive values (indicated by x) and variable references (**ptr** v), where **ptr** is a syntactic type annotation.

Expressions are given by the set *Expr*. Expressions $e \in Expr$ consist of numerals n, variables v, composite expressions $e \oplus e$ (where $\oplus$ is a binary operator), and operator **addressof** on program variables.

Commands $c \in Comm$ (where *Comm* is the set of the language commands) consist of atomic commands `skip`, variable definitions ($\texttt{def}(v)$), assignments ($v := e$), annotated assignments ($\langle v := e\rangle$), conditionals ($\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$), loops ($\texttt{while}\ e\ \texttt{do}\ c$) and sequential composition of commands ($c_1;\ c_2$).

$$
\begin{array}{rcl}
v & ::= & x \mid \mathbf{ptr}\ v \\
e & ::= & n \mid v \mid e \oplus e \mid \mathbf{addressof}(v) \\
c & ::= & \texttt{skip} \mid \texttt{def}(v) \mid v := e \mid \langle v := e\rangle \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid c_1\ ;\ c_2 \\
P & ::= & c
\end{array}
$$

Fig. 1. Abstract language.