Automatic Inference of Information Flow Properties

Bachelor's Thesis Description

Supervised by Marco Eilers and Jérôme Dohrau
ETH Zürich

Christian Knabenhans

February 2018

1 Motivation

A program has secure information flow if it does not leak any information about secret values like passwords or encryption keys in its output values. Non-interference is a program property which is closely related: Put simply, if all input and output variables of a given program are classified as either high (secret) or low (public), non-interference means that the values of the low outputs are not influenced by the high inputs, i.e., if the program is run multiple times with the same low inputs (but possibly different high inputs), the low outputs will be the same.

Information flow security is a hyperproperty, that is, it relates two executions of the same program. To prove such hyperproperties, we can build a product program that simulates two interleaved runs of the original program. We can then express the hyperproperty as a trace property of the product program, and verify it using off-the-shelf verifiers.

As part of the VerifiedSCION[1] project, such a technique for verifying secure information flow has been developed and implemented in [2], using the Viper [3] infrastructure. The method described in [2] allows one to specify hyperproperties in procedures specification (thus, it can reason about calls modularly), supports declassification (the programmer can explicitly allow some information to be leaked) and is termination- and time-sensitive (the analysis can prove that no information is leaked via the termination or non-termination, or the execution time of the program).

However, as is generally the case with deductive verification, this technique requires that specifications have to be added to the program in the form of pre- and postconditions as well as loop invariants, stating for example that certain inputs can be assumed to be low or high. Adding these specifications can mean a significant overhead for the programmer. Therefore, this thesis aims to develop a framework to automatically infer such specifications, in order to prove information flow security with less or even without programmer input.

Static analyses for secure information flow already exist, but they have limitations. The following function illustrates such a limitation:

[1] `http://www.pm.inf.ethz.ch/research/verifiedscion.html`

```
method f(public, secret: Int) returns (res: Int) {
        // secret is high, public and res are low
        res := public + secret;
        res := res - secret;
}
```

A simple static analysis that keeps track of which variables have been "tainted" by the high input variables (i.e., can potentially contain information about them) will fail to prove that no information about `secret` is leaked by `res`. However, if we use a more complex, relational analysis on the (simplified) product program `fxf` shown below, we will be able to prove secure information flow.

```
method fxf(public1, public2, secret1, secret2: Int) returns (res1, res2: Int)
{
        res1 := public1 + secret1
        res2 := public2 + secret2
        res1 := res1 - secret1
        res2 := res2 - secret2
}
```

To prove that no secret information is leaked, we need to prove that the output variables `res1` and `res2` will be equal if the low input variables `public1` and `public2` are equal. Using a relational abstract domain capable of expressing relationships between three variables, we can infer that just before the method returns, $\texttt{res1} = \texttt{public1} + \texttt{secret1} - \texttt{secret1} = \texttt{public1}$, and similarly for `res2`. From this information, we can automatically infer what we wanted to prove, namely $\texttt{public1} = \texttt{public2} \Rightarrow \texttt{res1} = \texttt{res2}$. However, such a simple relational analysis on product programs is also limited, and will for example not be able to prove that the following method does not leak any secret information:

```
method g(public: Int) returns (res: Int) {
        // both public and res are low
        if (public >= 0) { res := 1 }
        else { res := 2 }
}
```

It is obvious that `g`'s information flow is secure, since it does not have any high input variable. The (modular) product program of `g` is shown below. `a1` and `a2` are so-called activation variables: they allow to reason modularly about a program with many methods. Concretely, if the parameter `ai` is false, then the method has not been called in the `i`-th (simulated) call, and the method should not have any effect on the variables of the `i`-th call.

```
method gxg(a1, a2: Bool, public1, public2: Int) returns (res1, res2: Int) {
        var t1 := a1 && public1 >= 0
        var t2 := a2 && public2 >= 0
        var f1 := a1 && public1 < 0
        var f2 := a2 && public2 < 0
        if (t1) { res1 := 1 }
        if (t2) { res2 := 1 }
        if (f1) { res1 := 2 }
        if (f2) { res2 := 2 }
}
```

If the relational analysis from above is applied to the product program `gxg`, we will learn that $1 \leq \texttt{res1}, \texttt{res2} \leq 2$ before the method returns. The static analysis will not learn any information about the relation between `public1`, `public2` and `res1`, `res2`, and thus cannot infer that $\texttt{public1} = \texttt{public2} \Rightarrow \texttt{res1} = \texttt{res2}$. We hope that by using (variations or combinations of) standard analyses on modular product programs we can be more precise or more general than existing information flow analyses such as the ones presented above.

Sample[2] (Static Analyzer of Multiple Programming LanguageEs) is a generic static analyzer based on abstract interpretation [1] developed by the Chair of Programming Methodology. Abstract Interpretation provides mathematical guarantees about the state of program variables at any program point, and therefore is a suitable instrument to infer the desired specifications. Sample allows one to specify custom abstract domain and transformers.

2 Core Goals

- Apply static analysis with various abstract domains on programs found in the literature and their product programs. The goal of this step is to get a better understanding of the limitations of existing abstract domains and to find characteristics of product programs that could be exploited later in the project.
- Develop a static analysis that infers information flow properties. This includes exploring, combining, creating or possibly adapting existing abstract interpretation domains to solve problems found during the first phase of the project.
- Implement the static analysis in Sample by extending its semantic analysis component (abstract domain and transformers).
- Implement the inference in Sample. In this step we will use the result of the static analysis to infer secure information flow specifications about the program by extending the contract inference and the property checker parts of Sample.
- Test the implementation by inferring and verifying contracts for a number of examples in the Viper language from the literature. We also want to compare the precision and performance of our analysis with state-of-the-art tools, for example JOANA.[3]

3 Extension Goals

- Extend the static analysis to handle concurrent programs.
- Extend the static analysis to include more complex language constructs (e.g., arrays).
- Extend the static analysis to make it timing- or termination-sensitive.
- Extend the static analysis to other hyperproperties (e.g., determinism)

[2] `http://www.pm.inf.ethz.ch/research/sample.html`

[3] JOANA (Java Object-sensitive ANAlysis) is an information flow control framework for Java developed at the Karlsruhe Institute of Technology: `https://pp.ipd.kit.edu/projects/joana/`

References

[1] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[2] M. Eilers, P. Müller, and S. Hitz. "Modular Product Programs". In: *European Symposium on Programming (ESOP)*. LNCS. To appear. Springer-Verlag, 2018.

[3] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.