SAILS: Static Analysis of Information Leakage with Sample

Matteo Zanioli
Università Ca' Foscari, Venice, Italy
École Normale Supérieure, Paris, France
zanioli@dsi.unive.it

Pietro Ferrara
ETH, Zurich, Switzerland
pietro.ferrara@inf.ethz.ch

Agostino Cortesi
Università Ca' Foscari, Venice, Italy
cortesi@unive.it

ABSTRACT

In this paper, we introduce Sails, a new tool that combines Sample, a generic static analyzer, and a sophisticated domain for leakage analysis. This tool does not require to modify the original language, since it works with mainstream languages like Java, and it does not require any manual annotation. Sails can combine the information leakage analysis with different heap abstractions, inferring information leakage over programs dealing with complex data structures. We applied Sails to the analysis of the SecuriBench-micro suite. The experimental results show the effectiveness of our approach.

1. INTRODUCTION

Protecting the confidentiality of information stored in a computer system or transmitted over a public network is a relevant problem in computer security. The aim of information flow analysis is to prove the absence of leaks of sensitive information. Normally, there is an information flow from x to y whenever the information stored in x is transferred to, or used to derive information transferred to, y. Two kinds of information flow exist: explicit flow, when there is a direct flow between two variables (e.g., $\mathsf{y} = \mathsf{x}$), and implicit flow, when a statement specifies an explicit flow from z to y, but the execution depends on the value of a third variable x (e.g., $\mathsf{if}(\mathsf{x} > 0)\ \mathsf{y} = \mathsf{z};$). The starting point in secure information flow analysis is the classification of program variables into different security levels. In the simplest case, two levels are used: public (or low) and secret (or high). The main purpose is to prevent leak of sensitive information from a high variable to a low one. More generally, we might work with a lattice of security levels, ensuring that sensitive information flows only upwards in the lattice [11].

Language-based information flow security has been longly studied during the last decades [27, 26]. Proving that a program enforces noninterference has been the goal of several static analyses [15, 17, 5]. Nevertheless, despite this deep and extensive work, its practical applications have been relatively poor. Usually these approaches work on an ad-hoc programming language [3], and they do not support mainstream languages. This means that one should completely rewrite a program in order to apply it to some existing code.

Generally, works on information flow fall into two categories: dynamic, instrumentation-based approaches such as tainting, and static, language-based approaches such as type systems. The disadvantage of the dynamic approaches is that they typically incur significant run-time overhead [6, 20]. The disadvantage of the static approaches is that they typically require some changes to the language and the runtime environment, as well as non-trivial type annotations [24], making the adoption of these approaches difficult in practice.

On the one hand, Zanioli and Cortesi recently introduced a novel abstract interpretation-based information flow analysis [28]. This approach combines an information flow analysis with a numerical abstract domain. On the other hand, Ferrara developed a new generic static analyzer (Sample) based on abstract interpretation. Sample has been already applied to a wide range of different analyses (namely, string values [8], type information [13], and inference of access permissions [14]).

In this paper, we present Sails (Static Analysis of Information Leakage with Sample), an extension of Sample[1] to information leakage analysis. We slightly modify the theoretical approach to information flow analysis, presented in [28], in order to analyze object oriented programs using different heap abstractions (e.g., shape analysis [25, 16]), and some of the most powerful numerical abstract domains [18]. Unlike other works, our tool provides an information flow analysis without any changes to the language, since it tracks information flows between variables and heap locations over programs written in mainstream object-oriented languages like Java and Scala. We tested Sails over a set of web applications established as security and performance benchmarks. The experimental results show that the analysis is fast and effective in most of the code we analyzed.

The rest of this paper is organized as follows. Section 2 introduces the background of the existing information flow analysis and of Sample. Section 3 presents the main issues we solved in order to put together the two components. Section 4 shows the results of the analysis when applied to a pro-

[1] http://www.pm.inf.ethz.ch/research/semper/Sample

gram dealing with disjoint recursive data structures, while Section 5 discusses the experimental results when applying Sails to the SecuriBench-micro suite. Related work is presented in Section 6, while Section 7 concludes and depicts future works.

2. BACKGROUND

This section introduces some background about the information flow analysis adopted in Sails and about Sample.

2.1 Information Leakage Analysis

Zanioli and Cortesi [28] presented an information flow analysis by abstract interpretation. It combines a syntactic variable dependency analysis, based on a propositional formulae domain, with a variable value dependency using a numerical abstract domain. It uses logic formulae to represent dependencies between variables, refine the analysis in order to reduce as much as possible false alarms through information about numerical values, and detect information leakages evaluating formulae on truth-assignment functions. The resulting analysis has a modular construction that allows tuning the granularity of the abstraction, and the complexity of the abstract operators choosing a "good" compromise between efficiency and accuracy.

Consider, for example, the statement $\mathsf{if(x > 0)\ y = z;}$. The analysis adopts positive formulae, a subset of propositional formulae, to track both explicit (between y and z) and implicit (between x and y) flows. Formally, let $\Gamma = \{\wedge, \vee, \rightarrow, \neg\}$ be a set of connectives, $\Omega(\Gamma)$ be the set of formulae using the connectives in Γ, $\mathsf{V_p}$ the set of propositional variables, and $u : \mathsf{V_p} \rightarrow \mathtt{T}$ be a truth-assignment function which assigns to each variable the value *true*. Then the set of positive formulae is defined by: $\mathsf{Pos} = \{\mathsf{f} \in \Omega(\Gamma) \mid \mathsf{u} \vDash \mathsf{f}\}$, as in [7]. Roughly speaking, a propositional formulae is a Pos when, if you assign to each variables in the formulae the value T (*true*), the propositional formulae is satisfied. Some obvious examples are $\mathtt{T}, x_1 \in \mathsf{Pos}$ and $\mathtt{F}, \neg x_1 \notin \mathsf{Pos}$.

An abstract state $\overline{\sigma}^\sharp \in \overline{\Sigma}^\sharp \equiv \mathsf{L} \times \mathsf{Pos}$ (where L is the set of program labels) is a pair $\langle \ell, \phi \rangle$ which denotes the dependencies that occur among program variables up to label ℓ expressed by the positive formula $\phi \in \mathsf{Pos}$. Obviously, the propositional variables of Pos formulae, in this case, will be the program variables. To better understand, consider the command presented above. The analysis provides, at the end of the computation, the following propositional formula: $(z \rightarrow y) \wedge (x \rightarrow y)$.

The authors defined the abstract semantics as the set of all finite sets of abstract states, denoted by $\overline{\Sigma}^{\star\sharp}$, which can occur during one or more executions, in a finite time and starting from an initial state. The abstract domain is the lattice $\langle \overline{\Sigma}^{\star\sharp}, \sqsubseteq^\sharp, \emptyset, \overline{\Sigma}^\sharp, \sqcap^\sharp, \sqcup^\sharp \rangle$.

To improve the results, the analysis combines the abstract domains $\langle \overline{\Sigma}^{\star\sharp}, \sqsubseteq^\sharp, \emptyset, \overline{\Sigma}^{\star\sharp}, \sqcap^\sharp, \sqcup^\sharp \rangle$ and $\langle \mathsf{P}_n, \subseteq, \emptyset, \mathsf{R}^n, \uplus, \cap \rangle$ by a reduced product operator [10], where $\langle \mathsf{P}_n, \subseteq, \emptyset, \mathsf{R}^n, \uplus, \cap \rangle$ is the numerical abstract domain. The reduction is aimed at excluding pointless dependencies for all variables which have constant values during different executions, without loosing purposeful relations.

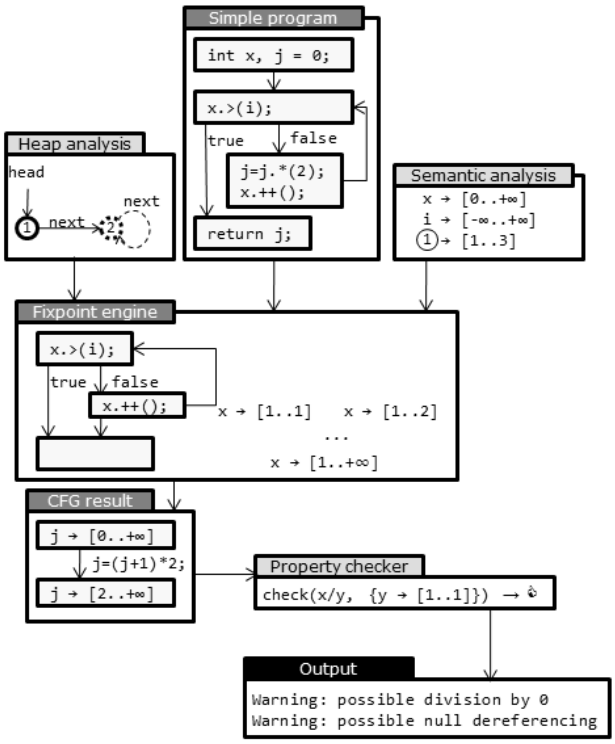

Figure 1: The structure of Sample

2.2 Sample

Sample (Static Analyzer of Multiple Programming LanguagEs) is a novel generic analyzer based on the abstract interpretation theory. This theory allows one to define compositional analyses [9]. Sample can be composed with different heap abstractions, approximations of other semantic information (e.g., numeric domains or information flow), properties of interest, and languages. Several heap analyses, semantic and numerical domains have been already plugged. The analyzer works on an intermediate language called Simple. Up to now, Sample supports the compilation of Scala and Java bytecode to Simple.

Picture 1 depicts the overall structure of Sample. Source code programs are compiled to Simple. A fixpoint engine receives a heap analysis, a semantic domain, and a program, and it produces an abstract result over a control flow graph for each method. This result is passed to a property checker that produces some output (e.g., warnings) to the user. Integrating an analysis in Sample allows one to take advantage of all aspects not strictly related to the analysis but that can improve its final precision (e.g., heap or numerical abstractions).

3. INTEGRATION OF THE ANALYSES

In this section, we present the main issues we have to deal with in order to combine the information leakage analysis with Sample.

3.1 Representing Propositional Formulae

To work with object oriented languages entailed to introduce some slight modifications on the domain for information leakage analysis described in [28]. We can consider a propositional formula ϕ as a conjunction of subformulae $(\zeta_0 \wedge \ldots \wedge \zeta_n)$. In the implementation, each subformula is an implication between two identifiers (an identifier is a variable abstraction, see Section 3.2). Then we represent a subformula as a pair of identifiers and a formula as a set of subformulae. Consider the simple example presented at the end of Section 2.1; the formula obtained after the analysis consists in two pairs: $(\overline{y}, \overline{z})$ and $(\overline{x}, \overline{y})$, where by $\overline{u}$ we denote the identifier of the variable u. The order relation "$\trianglelefteq$" is defined by: let ϕ_0 and ϕ_1 be propositional formulae, $\phi_0 \trianglelefteq \phi_1$ is equivalent to $\phi_0 \subseteq \phi_1$, where "$\subseteq$" is the classical subset relation.
Consequently, in the new abstract domain (PosDomain), the set of propositional variables ($\mathsf{V_p}$) consists in the set of identifier (Id), a propositional formulae (Pos) is represented by $\wp(\mathsf{Id} \times \mathsf{Id})$ and an abstract state $\overline{\sigma}^\sharp \in \overline{\Sigma}^\sharp$ is a propositional formula (Pos).

3.2 Heap Abstraction

In Sample heap locations are approximated by abstract heap identifiers. While the identifiers of program variables are fixed and represents exactly one concrete variable, the abstract heap identifiers may represent several concrete heap locations (e.g., if they summarize a potentially unbounded list), and they can be merged and split during the analysis. In particular we have to support (i) assignments on summary heap identifiers, and (ii) renaming of identifiers.

In order to preserve the soundness of Sails, we have to perform weak assignments to summary heap identifiers. Since a summary abstract identifier may represent several concrete heap locations and only one of them would be assigned in one particular execution, we have to take the upper bound between the assigned value, and the old one.

Any heap abstraction requires to rename, summarize or split existing identifiers. This information is passed through a replacement function $rep : \wp(\mathsf{Id}) \rightarrow \wp(\mathsf{Id})$. In TVLA [25] two abstract nodes represented by identifiers a_1 and a_2 may be merged to a summary node a_3, or a summary abstract node b_1 may be splitted to b_2 and b_3. Our heap analysis will pass $\{a_1, a_2\} \mapsto \{a_3\}$ and $\{b_1\} \mapsto \{b_2, b_3\}$ to Sails respectively. Given a single replacement $\mathsf{S}_1 \mapsto \mathsf{S}_2$, Sails removes all subformulae dealing with some of the variables in S_1, and for each removed subformula s it inserts a new subformula s' in the resulting state renaming each of the variables in S_1 to with each of the variables in S_2. Formally:

$$
\begin{array}{l}
rename : (\mathsf{Pos} \times (\wp(\mathsf{Id}) \rightarrow \wp(\mathsf{Id}))) \rightarrow \mathsf{Pos} \\
rename(\overline{\sigma}, rep) = \{(\mathsf{i'}_1, \mathsf{i'}_2) : (\mathsf{i}_1, \mathsf{i}_2) \in \overline{\sigma} \wedge \\
\quad \mathsf{i'}_1 = \begin{cases} \mathsf{i}_1 & \text{if } \nexists \mathsf{R}_1 \in dom(rep) : \mathsf{i}_1 \in \mathsf{R}_1 \\ \mathsf{k}_1 & \text{if } \exists \mathsf{R}_1 \in dom(rep) : \mathsf{i}_1 \in \mathsf{R}_1 \wedge \mathsf{k}_1 \in rep(\mathsf{R}_1) \end{cases}, \\
\quad \mathsf{i'}_2 = \begin{cases} \mathsf{i}_2 & \text{if } \nexists \mathsf{R}_2 \in dom(rep) : \mathsf{i}_2 \in \mathsf{R}_2 \\ \mathsf{k}_2 & \text{if } \exists \mathsf{R}_2 \in dom(rep) : \mathsf{i}_2 \in \mathsf{R}_2 \wedge \mathsf{k}_2 \in rep(\mathsf{R}_2) \end{cases} \}
\end{array}
$$

3.3 Implicit Flow Detection

An implicit information flow occurs when there is an information leakage from a variable in a condition to a variable assigned inside a block dependent on that condition. For instance, in if(x > 0) y = z; there is an explicit flow from z to y, and an implicit flow from x to y. To record these relations we relate the variables in the conditions to the variables that have been assigned in the block. When we join two blocks coming from the same condition, we discharge all implicit flows on the abstract state.

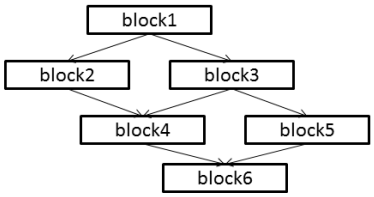

Figure 2: A CFG not supported by Sails

On the other hand, Sample programs are represented by control flow graphs (cfg), and therefore we could have conditions that do not join in a well-defined point. For instance, in the cfg of Figure 2 is not clear if the condition of block 1 is joined at block 4 or 6. For this reason, Sails does not support all cfgs that can be represented in Sample but only the ones coming from structured programs, i.e., that corresponds to programs with if and while statements and not with arbitrary jumps like goto.

3.4 Property

An information flow analysis can be carried out by considering different attacker abilities. We implemented two scenarios: when the attacker can read public variables only at the beginning and at the end of the computation, and when the attacker can read public variables after each step of the computation[2]. Moreover, to each attacker we implemented two security properties: secrecy (i.e., information leakage analysis) and integrity.
The verification of these properties is based on the following steps: computation of the analysis, declaration of private variables (at run time, by a text files writing the variables name or by a graphical user interface selecting the variables in a list) and verification of the property.

3.5 Numerical Analyses

The information flow analysis is based on the reduced product of a dependency and a numerical analysis. Thanks to the structure of Sample, we can naturally plug Sails with different numerical domains. In particular, Sample supports the Apron library [18]. In this way, we can combine Sails with all numerical domains contained in Apron (namely, Polka, the Parma Polyhedra Library, Octagons, and a deep implementation of Intervals).

In addition, we can apply different heap abstractions to the analysis of a program without changing Sails. For instance, if we are not interested to the heap structure, we can use a less accurate domain that approximates all heap locations with one unique summary node, as in Section 5. Instead, if we look at a precise abstraction of the heap structure, we

[2]Notice that, as in [28], we assume that the attacker, in both cases, knows the source code of the program.

```
1 class ListWorkers {
2   int salary;
3   ListWorkers next;
4   ...
5 }
6
7 public void updateSalaries(ListWorkers employees, ListWorkers managers) {
8   int maxSalary = 0;
9   ListWorkers it=employees;
10  while(it!=null) {
11    if(it.salary>maxSalary)
12      maxSalary=it.salary;
13    it=it.next;
14  }
15  it=managers;
16  while(it!=null) {
17    if(it.salary < maxSalary)
18      it.salary=maxSalary;
19    it=it.next;
20  }
21 }
```

Figure 3: A motivating example

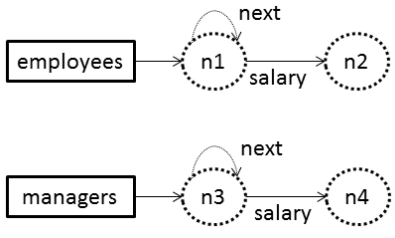

Figure 4: The initial state of the heap abstraction

can adopt more precise approximations, as illustrated in the next section.

4. EXAMPLE

Consider the Java code in Figure 3. Class ListWorkers models a list of workers of an enterprise. Each node contains the salary earned by the worker, and some other information (e.g., name and surname of the person).
Method updateSalaries is defined as well. It receives a list of employees and a list of managers. These two lists are supposed to be disjoint. First method updateSalaries computes the maximal salary of an employee. Then it traverses the list of managers updating their salary to the maximal salary of employees if manager's salary is smaller than that.

Usually managers would like not to leak information about their salary to employees. This property could be expressed in Sails specifying that we do not want to have a flow of information *from* managers *to* employees. More precisely, we want to prove the absence of information leakage from the content of field salary of any node reachable from managers to any node reachable from employees.

We combine Sails with a heap analysis that approximates all objects created by a program point with a single abstract node[12]. We start the analysis of method updateSalaries with an abstract heap in which lists managers and employees are abstracted with a summary node and they are disjoint. Figure 4 depicts the initial state, where n2 and n4 contains the salary values of the ListWorkers n1 and n3, respectively. In the graphic representation we adopt dotted circles to represent summary nodes, rectangles to represent local variables, and edges between nodes to represent what is pointed by fields of objects. Note that the structure of these two lists does not change during the analysis of the program, since method updateSalaries does not modify the heap structure.

Sails infers that, after the first while loop at line 15, there is a flow of information from n2 to maxSalary. This happens because variable it points to n1 before the loop (because of the assignment at line 9), and it iterates following field next (obtaining always the summary node n1) eventually assigning the content of it.salary (that is, node n2) to maxSalary. Therefore, at line 15 we have the propositional formula $n2 \rightarrow maxSalary$.

Then updateSalaries traverses list managers. For each node, it could potentially assign maxSalary to it.salary. Similarly to what happened in the previous loop, variable it points to n3 before and inside the loop, since field next always points to the summary node n3. Therefore the assignment at line 18 could potentially affects only node n4. For this reason, Sails discovers a flow of information from maxSalary to n4, represented by the propositional formula $maxSalary \rightarrow n4$.

At the end of the analysis, Sails soundly computes that $(n2 \rightarrow maxSalary) \wedge (maxSalary \rightarrow n4)$. By the transitive property, we know that there could be a flow of information from n2 to n4, that is, from employees to managers. This flow is allowed by our security policy. On the other hand, we also discovered that there is no information leakage from list managers to list employees, since Sails does not contain any propositional formula containing this flow. Therefore Sails proves that this program is safe.

Notice that, almost 10 years ago, Sabelfeld and Myers stated: "Noninterference of programs essentially means that a *variable* of confidential (high) input does not cause a variation of public (low) output"[24]. Thanks to the combination between a heap abstraction and an abstract domain tracking information flow, Sails deals directly with the structure of the heap, extending the concept of noninterference from variables to portions of the heap represented by abstract nodes. This opens a new scenario since we can prove that a whole data structure does not interfere with another one, as we have done in this example. As far as we know, Sails is the only tool that performs a noninterference analysis over a heap abstraction, and therefore it can prove properties like "there is no information flow from the nodes reachable from v_1 to the nodes reachable from v_2".

5. EXPERIMENTAL RESULTS

A well-established way of studying the precision and the efficiency of information flow analyses is the SecuriBench-micro suite [2], a set of small test cases designed to verify different parts of static security analyzer. We applied Sails to this test suite; the description and the results of these benchmarks are reported in Table 1. Column `fa` reports if the analysis did not produce any false alarm. We combined Sails with a really rough heap abstraction that approximates all concrete heap locations with one abstract node. Sails detected all information leakages in all tests, but in three

Name	Description	fa
Aliasing1	Simple aliasing	✓
Aliasing2	Aliasing false positive	✓
Basic1	Very simple XSS	✓
Basic2	XSS combined with a conditional	✓
Basic3	Simple derived integer test	✓
Basic5	Test of derived integer	✓
Basic6	Complex test of derived integer	✓
Basic8	Test of complex conditionals	✓
Basic9	Chains of value assignments	✓
Basic10	Chains of value assignments	✓
Basic11	A simple false positive	✓
Basic12	A simple conditional	✓
Basic18	Protect agains simple loop unrolling	✓
Basic28	Complicated control flow	✓
Pred1	Simple if(false) test	✗
Pred2	Simple correlated tests	✓
Pred3	Simple correlated tests	✓
Pred4	Test with an integer variable	✓
Pred5	Test with a complex conditional	✓
Pred6	Test with addition	✗
Pred7	Test with multiple variables	✗

Table 1: SecuriBench-micro suite

Name	Description	fa
A	Simple explicit flow test	✓
Account	Simple explicit flow test	✓
ConditionalLeak	Explicit flow in if statement	✓
Do	Implicit flow in the loop	✓
Do2	Implicit flow if and loop	✓
Do3	Implicit flow loop and if	✓
Do4	Implicit flow loop and if	✓
Do5	Implicit flow loop and if	✓
If1	Simple implicit flow	✓
Implicit	Simple implicit flow	✓

Table 2: Jif case studies

cases (Pred1, Pred6 and Pred7) it produced false alarms. This happens because Sails abstracts away the information produced when testing to true or false boolean conditions in if or while statements. We are currently investigating how to extend the analysis with more complex propositional formulae to avoid this kind of false alarms.

Since these benchmarks cover only problems with explicit flows, we performed further experiments using some Jif [22] case studies. The results are reported in Table 2: we discovered all flows without producing any false alarm. These results allow us to conclude that Sails is precise, since in 90% of the cases (28 out of 31 programs) it does not produce any false alarm.

About the performances, the analysis of all case studies takes 1.092 seconds (0.035 sec per method in average) without combining it with a numerical domain. When we combine it with Intervals it takes 3.015 seconds, whereas it takes 6.130 seconds in combination with Polka. All tests are performed by a MacBook Pro, Intel Core 2 Duo 2.53 GHz, 4 GB Memory. Therefore the experimental results underline the efficiency of Sails as well.

6. RELATED WORK

The approach adopted in Sails is quite different from existing tools that deal with information flow analysis. The most known tool in this field is Jif [3]. It is a security-typed programming language that extends Java with support for information flow and access control, enforced at compile time. Jif is an ad hoc analysis that requires to annotate the code with some type information. If on the one hand Jif is more efficient than Sails, on the other hand Sails does not require any manual annotation, and it takes all advantages of compositional analyzers (e.g., we can combine Sails with a TVLA-based heap abstraction).

Other security-typed languages emerged over the years to prevent insecure information flows. The possibility of regulating the propagation of sensitive information by security type systems in realistic languages came, in the early 2000, from [4, 21, 23] and their implementations.

"Despite this rather large (and growing!) body of work on language-based information-flow security, there has been relatively little adoption of the proposed techniques"[19]. According to Li and Zdancewic, one of the reasons that limited the application of these systems is that they require to rewrite the whole system in the new language. In addition, usually only a small part of the system deals with critical information. Therefore developers choose the programming language that best fits the primary functionality of the system.

Our approach does not require to change the programming language, since it infers the flow of information directly on the original program, and it asks what are the private data that have not to be leaked to the user during the analysis execution.

7. CONCLUSIONS

In this paper we presented Sails, a new tool that performs static analysis of information flow over object-oriented programs. Sails represents the combination of Sample (a generic static analyzer) and an information leakage analysis [28]. Thanks to this combination, Sails is modular w.r.t. the heap abstraction, and it can verify noninterference over recursive data structures using simple and efficient heap analyses. In addition, it can be combined with several numerical domains. The experimental results underline the effectiveness of the analysis, since Sails is in position to analyze several benchmarks in about 1 seconds without producing false alarms in more than 90% of the programs.

7.1 Future Work

Sample interprocedural semantics relies on contracts, but it does not yet support contracts dealing with levels of confidentiality. Then we are working to extend the annotation language to define contracts like "x.f is confidential ". Once we will have defined this language, we will apply Sails to bigger (and hopefully industrial) case studies like the ones contained in SecuriBench [1].
In addition, a plugin to interface TVLA with Sample has been developed recently [16]. A more sophisticated shape analysis that avoids the summarization of nodes with different level of confidentiality may in fact enhance the precision of the Sails analysis.

Aknowledgments

Work partially supported by RAS project "TESLA - Tecniche di enforcement per la sicurezza dei linguaggi e delle applicazioni" and by SNF project "Verification-Driven Inference of Contracts".

8. REFERENCES

[1] Stanford SecuriBench. http://suif.stanford.edu/~livshits/work/securibench/.

[2] Stanford SecuriBench Micro. http://suif.stanford.edu/~livshits/work/securibench-micro/.

[3] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *ESORICS*, Lecture Notes in Computer Science, pages 197–221, 2005.

[4] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 253–267. IEEE Computer Society Press, 2002.

[5] C. Braghin, A. Cortesi, and F. Focardi. Information flow security in boundary ambients. *Information and Computation*, 206(2-4):460–489, 2008.

[6] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[7] A. Cortesi, G. File', and W. Winsborough. Optimal groundness analysis using propositional logic. *The Journal of Logic Programming*, 27(2):137 – 167, 1996.

[8] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011)*, Lecture Notes in Computer Science. Springer, October 2011.

[9] P. Cousot. The Calculational Design of a Generic Abstract Interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[11] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[12] P. Ferrara. A fast and precise alias analysis for data race detection. In *Proceedings of the Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science. Elsevier, April 2008.

[13] P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 6117 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2010.

[14] P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science. Springer, January 2012.

[15] R. Focardi and M. Centenaro. Information flow security of multi-threaded distributed programs. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 113–124, New York, NY, USA, 2008. ACM.

[16] R. Fuchs. Interfacing tvla and sample. Bachelor thesis, ETH Zurich, August 2011.

[17] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 186–197, New York, NY, USA, 2004. ACM.

[18] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV 2009)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

[19] P. Li and S. Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411:1974–1994, April 2010.

[20] Y. Liu and A. Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of 14th European Conference on Software Maintenance and Reengineering*, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[21] A. C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[22] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. software release., July 2001-2004.

[23] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25:117–158, January 2003.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan. 2003.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[26] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, New York, NY, USA, 1998. ACM.

[27] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[28] M. Zanioli and A. Cortesi. Information leakage analysis by abstract interpretation. In *Proceedings of the 37th international conference on Current trends in theory and practice of computer science*, volume 6543 of *Lecture Notes in Computer Science*, pages 545–557. Springer-Verlag, 2011.