Noninterference for Mobile Code with Dynamic Security Domains

Robert Grabowski[1]

Computer Science Department, University of Munich, Germany

Abstract

Language-based information flow analysis is used to statically examine a program for information flows between objects of different security domains, and to verify these flows follow a given policy.
When the program is distributed as mobile code and executed in different environments, the program may access external objects whose domains are not known until runtime. To maintain information flow security, runtime tests are required to determine which flows between accessed objects are actually allowed before operations inducing these flows are performed.
We present an imperative language with heaps, domains as values and classes with restricted dependent types. We use a type system to statically prove that the flow tests included in a program are sufficient, such that a noninterference property for the data objects is ensured regardless of their domains.

Keywords: Language-based information flow security, noninterference, dynamic security domains

1 Introduction

The goal of information flow security is to control the flow of information between data objects of a computing system, like variables, files, or sockets. More precisely, each object is assigned a *security domain*. An *information flow policy* defines the allowed flows between objects of these domains. A way to verify this property for a given program is to perform a static analysis prior to the execution, checking whether a program transfers data between those objects in a manner that respects the policy [7]. Programs with this property are also called *noninterferent*.

While this language-based approach has evolved into a larger research field [11], most works assume the objects a program uses and their domains can be statically inferred. In mobile code scenarios, however, the program is executed in different client environments, where the set of available data objects and their domains typically differ.

For example, consider the program in figure 1 that copies the contents of one file via a temporary variable to another. The security of this program depends on the domain of the contents of the first and the second file. Additionally, there are

---

[1] Email: robert.grabowski@ifi.lmu.de

```
file1 := openFile(filename1);
file2 := openFile(filename2);
tmp := readFile(file1);
writeFile(file2, tmp);
closeFile(file1);
closeFile(file2);
```

Fig. 1. Simple file copy program

indirect flows: a change to the file contents reveals information about the filename variables, and the (non-)existence of files may be uncovered by a triggered file-not-found exception.

To keep things simple, we avoid indirect flows by assuming a fixed set of files and a fixed file selection for the time being. Each file contents may have an arbitrary *dynamic domain* which is not known before runtime. Our goal is to ensure that a program manipulating the files is secure for *any* domains the files may have.

In this paper, we present a small imperative language with a variable store that may contain references to an extendable heap. The heap contains simple objects, i.e. structures with data fields. As is standard, we assume there are only two domains L and H for low and high security data. A typing environment assigns a domain to each variable and object field. Data may not flow from variables or fields of domain H to variables or fields of domain L.

The main extension is that we allow a very restricted form of dependent types for objects: each object has a special immutable field δ whose value can determine the domain of other fields. A file with a dynamic domain, for example, could then be modeled as an object of the class `File` with fields typed according to the following pseudo-specification:

```
class File {
    δ : L;
    contents : this.δ;
}
```

The domain field δ can be inspected and compared during runtime. The programmer can use this feature to dynamically check whether a flow between data objects is permitted by the policy before performing an action that induces such a flow. For example, if f_1 and f_2 are references to `File` objects, a check that secures the file copy operation may look as follows:

if $f_1.\delta \sqsubseteq f_2.\delta$ **then**
$\quad f_2.contents := f_1.contents$

We present a static analysis in form of a type system in the style of the system by Volpano, Smith, and Irvine [13]. Since the actual domains of fields with type *this.δ* are not statically available, the analysis is performed symbolically on the domain fields, checking whether each flow between these domains is guarded by an enclosing runtime test. The security notion for the program implies a noninterference result for the objects, *regardless* of the value of the δ field, i.e. the domain of their contents.

The language is also suitable to write programs for code consumer environments

where even the security policy is not known before run-time. The label check shown above is also a dynamic test of the policy currently in effect. If we drop any assumptions about the policy in the analysis, a well-typed program can be proven secure for any given information flow policy.

Related work

Lantian Zheng and Andrew Myers [14] propose a functional language λ_{DSec} with security types in the style of the SLam calculus [8]. Their language features dynamic domains that can be used as values, and a constrained form of dependent pair and function types. It is used to formalise dynamic label mechanisms in Jif [9].

Our language is a WHILE language with heaps and objects, extended towards dynamic domains as featured in λ_{DSec}. While additional features such as higher-order functions and heaps make λ_{DSec} more expressive than our language, choosing an object-oriented imperative language makes it possible to build on existing results by Barthe et al. for certified compilation to bytecode that preserves security types [2,3,4]. This provides a good starting point for embedding dynamic domains into a proof-carrying code setting [10]. Also, the language allows for successively identifying fragments of λ_{DSec} that are suitable for certified compilation to flow-secure bytecode.

Main contributions

This paper makes the following contributions:

- an imperative language with dynamic security domains (section 2),
- simple objects with restricted dependent types as a programming model for external objects with dynamic domains, and a generalised noninterference property for them (section 3), and
- a sound type system with symbolic evaluation of the domain variable order (section 4)

The paper concludes with a discussion of compilation to bytecode and further benefits of the language in mobile code scenarios, and provides an outlook towards future research.

2 Language

2.1 Preliminaries

The policy is modeled as a lattice over a set of domains $\mathcal{D} = \{L, H\}$, where L is the bottom and H the top element. The reflexive and transitive order $\sqsubseteq$ represents the permitted information flows; we assume $L \sqsubseteq H$ and $H \not\sqsubseteq L$. The operators $\sqcup$ and $\sqcap$ compute the least upper bound and greatest lower bound of two domains.

Let x range over a set of variables $\mathcal{X}$, m range over a set of memory locations $\mathcal{M}$, and f range over a set of fields $\mathcal{F}$. Let n stand for a natural number from $\mathbb{N}$, and k for a domain from $\mathcal{D}$. The table in figure 2 shows the program state model.

A state σ is a pair (s, h) of a store and a heap. A store is a variable assignment,

$$\text{references: } r \in \mathcal{R} ::= null \mid m \qquad \text{heaps: } h \in \mathcal{H} = \mathcal{M} \rightharpoonup \mathcal{O}$$

$$\text{values: } v \in \mathcal{V} ::= n \mid k \mid r \qquad \text{stores: } s \in \mathcal{S} = \mathcal{X} \to \mathcal{V}$$

$$\text{objects: } o \in \mathcal{O} = \mathcal{F} \rightharpoonup \mathcal{V} \qquad \text{states: } \sigma \in \Sigma = \mathcal{S} \times \mathcal{H}$$

Fig. 2. Program state model

i.e. a function from variables to values, while a heap assigns each memory location an object. Since a heap may be extended with new objects, it is a partial function. An object is a finite, partial mapping of field identifiers to values. The set $\mathcal{F}$ must contain a special identifier δ. We require $\delta \in \text{dom}(o)$ for all objects o. For functions, we use the update operation $f[x \mapsto y]$ to overwrite the value of x in f with y.

Note that we follow Barthe et al. [4] and call the record-like data structures on the heap "objects". In section 6, we briefly outline how to equip them with methods.

2.2 Syntax and semantics

Let C to range over a set of class identifiers. The language has the following syntax for expressions and programs:

$$e \in \mathcal{E} ::= v \mid x \mid x.f \mid e_1 \ \mathbf{op} \ e_2 \mid e_1 \sqcup e_2 \mid e_1 \sqsubseteq e_2$$

$$P \in \mathcal{P} ::= P_1 \ ; \ P_2 \mid \mathbf{if} \ e \ \mathbf{then} \ P_1 \ \mathbf{else} \ P_2 \mid \mathbf{while} \ e \ \mathbf{do} \ P \mid$$

$$\mathbf{skip} \mid x := e \mid x.f := e \mid x := \mathbf{new} \ C(e)$$

The syntax for expressions consists of literal values, variables, a field read access, and a binary operator, as well as a least upper bound operator and an order test for domains. The binary operator could be e.g. the addition of two numbers, a greater-than-comparison or another primitive operation. In any case, we assume its semantics to be given by a binary function $\mathrm{op} : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$. The table in figure 3 defines $[\![e]\!]_{s,h}$, the evaluation of an expression e in a program state (s, h). It is undefined if e.g. an accessed field does not exist or if the binary operators are undefined.

The syntax for programs forms a WHILE language with object creation and field access. The big-step operational semantics shown in figure 4 is standard; $\sigma_1 \xrightarrow{P} \sigma_2$ means "executing P in state σ_1 results in state σ_2". For branching expressions, we interpret natural numbers to avoid the need for explicit boolean values.

Following Barthe et al. [4], the semantics of $\mathbf{new} \ C(e)$ relies on a given function default_C which returns a default object for class C with the appropriate set of fields and default values. The class identifier C merely selects the default_C function to use, no explicit class information is stored on the heap. The expression e can be seen as a constructor argument: its value is used to initialise the δ field. This extension is needed since the type system ensures that the δ field is immutable.

$$\llbracket v \rrbracket_{s,h} = v \qquad\qquad \llbracket e_1 \textbf{ op } e_2 \rrbracket_{s,h} = \llbracket e_1 \rrbracket_{s,h} \text{ op } \llbracket e_2 \rrbracket_{s,h}$$

$$\llbracket x \rrbracket_{s,h} = s(x) \qquad\qquad \llbracket e_1 \sqcup e_2 \rrbracket_{s,h} = \llbracket e_1 \rrbracket_{s,h} \sqcup \llbracket e_2 \rrbracket_{s,h}$$

$$\llbracket x.f \rrbracket_{s,h} = h(s(x))(f) \qquad\qquad \llbracket e_1 \sqsubseteq e_2 \rrbracket_{s,h} = \llbracket e_1 \rrbracket_{s,h} \sqsubseteq \llbracket e_2 \rrbracket_{s,h}$$

Fig. 3. Denotational semantics of expressions

$$\text{SKIP } \frac{}{\sigma_1 \xrightarrow{\textbf{skip}} \sigma_1} \qquad\qquad \text{SEQ } \frac{\sigma_1 \xrightarrow{P_1} \sigma_2 \qquad \sigma_2 \xrightarrow{P_2} \sigma_3}{\sigma_1 \xrightarrow{P_1\ ;\ P_2} \sigma_3}$$

$$\text{IF-T } \frac{\llbracket e \rrbracket_{\sigma_1} > 0 \qquad \sigma_1 \xrightarrow{P_1} \sigma_2}{\sigma_1 \xrightarrow{\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2} \sigma_2} \qquad\qquad \text{IF-F } \frac{\llbracket e \rrbracket_{\sigma_1} = 0 \qquad \sigma_1 \xrightarrow{P_2} \sigma_2}{\sigma_1 \xrightarrow{\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2} \sigma_2}$$

$$\text{WHILE-T } \frac{\llbracket e \rrbracket_{\sigma_1} > 0 \qquad \sigma_1 \xrightarrow{P} \sigma_2 \qquad \sigma_2 \xrightarrow{\textbf{while } e \textbf{ do } P} \sigma_3}{\sigma_1 \xrightarrow{\textbf{while } e \textbf{ do } P} \sigma_3} \qquad \text{WHILE-F } \frac{\llbracket e \rrbracket_{\sigma_1} = 0}{\sigma_1 \xrightarrow{\textbf{while } e \textbf{ do } P} \sigma_1}$$

$$\text{ASSIGN } \frac{s_2 = s_1[x \mapsto \llbracket e \rrbracket_{s_1,h_1}]}{(s_1, h_1) \xrightarrow{x := e} (s_2, h_1)}$$

$$\text{PUTFIELD } \frac{s_1(x) = m \qquad h_1(m)[f \mapsto \llbracket e \rrbracket_{s_1,h_1}] = o \qquad h_2 = h_1[m \mapsto o]}{(s_1, h_1) \xrightarrow{x.f := e} (s_1, h_2)}$$

$$\text{NEW } \frac{\begin{array}{c} m \notin \text{dom}(h_1) \\ \llbracket e \rrbracket_{s_1,h_1} = k \qquad \text{default}_C \cup [\delta \mapsto k] = o \qquad h_2 = h_1 \cup [m \mapsto o] \end{array}}{(s_1, h_1) \xrightarrow{x := \textbf{new } C(e)} (s_1, h_2)}$$

Fig. 4. Operational semantics

Also note that fields can only be accessed directly via a reference variable. Programs with indirect field access such as $x.y.f$ can be rewritten using additional auxiliary variables. We do not handle invalid field accesses $x.f$; if f does not exist or x is not a reference, the semantics is simply not defined.

3 Noninterference

Noninterference is defined relative to an environment $\Gamma : \mathcal{X} \to \mathcal{D}$ that assigns a domain to each variable, and an environment $\mathsf{fieldlevel} : \mathcal{F} \to \mathcal{D} \cup \{this.\delta\}$ that assigns a domain to each field. A domain $this.\delta$ assigned to field f means the

domain of the field f is stored in the δ field of the same object. We require the domain variable itself to have low security, i.e. $\mathsf{fieldlevel}(\delta) = L$. (A high-typed field δ would not permit any interesting propositions about fields with type $this.\delta$.) Since Γ and $\mathsf{fieldlevel}$ remain fixed, we use them implicitly in the following.

A program P is considered secure if the values of L variables in the final state do not depend on the values of H variables in the initial state. We formalise this using the usual noninterference property: if two states are equivalent, i.e. indistinguishable on all L variables, then running P on both states must result in two equivalent states.

Since heap objects may have different allocations in both heaps, we follow the approach by Banerjee and Naumann [1] and use a partial bijection β that relates the memory locations of two objects that should be indistinguishable.

Value indistinguishability with respect to β is defined by the following clauses:

$$n \sim_\beta n \qquad\qquad k \sim_\beta k \qquad\qquad null \sim_\beta null \qquad\qquad \frac{\beta(m) = m'}{m \sim_\beta m'}$$

For two objects, $o \sim_\beta o'$ holds if $\mathrm{dom}(o) = \mathrm{dom}(o')$, and for all fields $f \in \mathrm{dom}(o)$,

- $\mathsf{fieldlevel}(f) = L$ implies $o(f) \sim_\beta o'(f)$
- $\mathsf{fieldlevel}(f) = this.\delta$ and $o.\delta = L$ implies $o(f) \sim_\beta o'(f)$

Note that $o.\delta = o'.\delta$ whenever $o \sim_\beta o'$, since $\mathsf{fieldlevel}(\delta) = L$. The relation is extended to stores, heaps and states as follows:

$$s \sim_\beta s' \Longleftrightarrow \forall x.\, \Gamma(x) = L \;\Rightarrow\; s(x) \sim_\beta s'(x)$$
$$h \sim_\beta h' \Longleftrightarrow \mathrm{dom}(\beta) \subseteq \mathrm{dom}(h) \;\wedge\; \mathrm{rng}(\beta) \subseteq \mathrm{dom}(h') \;\wedge$$
$$\forall m \in \mathrm{dom}(h).\, h(m) \sim_\beta h'(\beta(m))$$
$$(s,h) \sim_\beta (s',h') \Longleftrightarrow s \sim_\beta s' \;\wedge\; h \sim_\beta h'$$

For a noninterferent program, if the initial states are indistinguishable with respect to a bijection β, then the final states must be indistinguishable with respect to a bijection γ that is possibly larger, since new objects might have been created and bound to a low reference.

Definition 3.1 A program P is *noninterferent* if for all states $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$ and all partial bijections β it holds

$$\sigma_1 \sim_\beta \sigma'_1 \;\wedge\; \sigma_1 \xrightarrow{P} \sigma_2 \;\wedge\; \sigma'_1 \xrightarrow{P} \sigma'_2 \;\Rightarrow\; \sigma_2 \sim_\gamma \sigma'_2$$

for some partial bijection $\gamma \supseteq \beta$.

As noninterference is defined termination-insensitive here, a non-terminating program is trivially secure.

4 Type system

The type system is an extension of the system by Volpano, Smith, and Irvine [13]. The soundness result shows that if a program is well-typed, then it is noninterferent. The type system does not handle the orthogonal aspect of data type safety.

$$\overline{C \vdash \ell \leq \ell} \qquad \overline{C \vdash \ell_1 \sqcup \ell_2 \leq \ell_2 \sqcup \ell_1} \qquad \overline{C \vdash L \leq \ell} \qquad \overline{C \vdash \ell \leq H}$$

$$\frac{(\ell_1, \ell_2) \in C}{C \vdash \ell_1 \leq \ell_2} \qquad \frac{C \vdash \ell_1 \leq \ell_2 \quad C \vdash \ell_2 \leq \ell_3}{C \vdash \ell_1 \leq \ell_3} \qquad \frac{C \vdash \ell_1 \leq \ell_3 \quad C \vdash \ell_2 \leq \ell_3}{C \vdash \ell_1 \sqcup \ell_2 \leq \ell_3}$$

Fig. 5. Label order rules

$$\overline{C \vdash v \; : \; L} \qquad \overline{C \vdash x \; : \; \Gamma(x)} \qquad \overline{C \vdash x.f \; : \; \mathsf{fieldlevel}(f)[x/this] \sqcup \Gamma(x)}$$

$$\frac{\circ \in \{\mathbf{op}, \sqcup, \sqsubseteq\} \quad C \vdash e_1 \; : \; \ell \quad C \vdash e_2 \; : \; \ell}{C \vdash e_1 \circ e_2 \; : \; \ell} \qquad \frac{C \vdash e \; : \; \ell \quad C \vdash \ell \leq \ell'}{C \vdash e \; : \; \ell'}$$

Fig. 6. Expression typing rules

4.1 Labels

Since the concrete values (domains) of the domain variables are not known statically, the type system performs symbolic reasoning on an extension of domains called *labels*. Labels form a subset of expressions: the can be domain constants, a domain variable (i.e. a δ field), or a least upper bound of two labels:

$$\ell \in \mathcal{L} ::= k \mid x.\delta \mid \ell_1 \sqcup \ell_2$$

The type system provides a special analysis for label tests, i.e. conditional statements of the form **if** $\ell_1 \sqsubseteq \ell_2$ **then** P_1 **else** P_2. For the subprogram P_1, we can assume that an information flow from ℓ_1 to ℓ_2 is valid; otherwise, the branch will not be taken at runtime. The typing judgements for expressions and programs are therefore parametrized over a set $C \subseteq \mathcal{L} \times \mathcal{L}$ containing label pairs. A pair $(\ell_1, \ell_2) \in C$ expresses the assumption that a flow from ℓ_1 to ℓ_2 is allowed.

The set C thus stores abstract information about the δ fields of objects at a point of execution. Since C gives requirements for suitable program states, we also call it the *constraint set*. The following definition formalises the requirement:

Definition 4.1 A program state σ *satisfies* a set of contraints C, if for all pairs $(\ell, \ell') \in C$ it holds $[\![\ell]\!]_\sigma \sqsubseteq [\![\ell']\!]_\sigma$.

The label flow information in C induces an order on the labels, which is a superset of the domain order $\sqsubseteq$. With the rules in figure 5, we can derive a judgement of the form $C \vdash \ell_1 \leq \ell_2$, which states that information may flow from ℓ_1 to ℓ_2. The following theorem states the label order derived from C is sound with respect to their interpretation in satisfying program states.

Theorem 4.2 (Soundness of label order rules) *Given a set of constraints C, two labels ℓ and ℓ' and a state σ satisfying C, then $C \vdash \ell \leq \ell'$ implies $[\![\ell]\!]_\sigma \sqsubseteq [\![\ell']\!]_\sigma$.*

Proof. By induction over the derivation of the label ordering. $\qquad \square$

$$\text{(SUB)} \quad \frac{C, k' \vdash P \qquad C \vdash k \le k'}{C, k \vdash P} \qquad\qquad \text{(SKIP)} \quad \frac{}{C, H \vdash \mathbf{skip}}$$

$$\text{(COMP)} \quad \frac{C, k \vdash P_1 \qquad C, k \vdash P_2}{C, k \vdash P_1 \,;\, P_2} \qquad\qquad \text{(ASSIGN)} \quad \frac{\Gamma(x) = k \qquad C \vdash e : k \qquad x.\delta \notin C}{C, k \vdash x := e}$$

$$\text{(IF)} \quad \frac{C \vdash e \,:\, k \qquad C, k \vdash P_i \ \text{ for } i \in \{1, 2\}}{C, k \vdash \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2} \qquad\qquad \text{(WHILE)} \quad \frac{C \vdash e \,:\, k \qquad C, k \vdash P}{C, k \vdash \mathbf{while}\ e\ \mathbf{do}\ P}$$

$$\text{(IF-LABEL)} \quad \frac{C \vdash \ell_1 \,:\, k \qquad C \vdash \ell_2 \,:\, k \qquad C \cup \{(\ell_1, \ell_2)\}, k \vdash P_1 \qquad C, k \vdash P_2}{C, k \vdash \mathbf{if}\ \ell_1 \sqsubseteq \ell_2\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2}$$

$$\text{(PUT-FIELD)} \quad \frac{f \ne \delta \atop \mathsf{fieldlevel}(f)[x/this] = \ell \qquad C \vdash e \,:\, \ell \qquad C \vdash \Gamma(x) \le \ell \qquad C \vdash k \le \ell}{C, k \vdash x.f := e}$$

$$\text{(NEW)} \quad \frac{C \vdash e \,:\, L \qquad \Gamma(x) = k \qquad x \notin C}{C, k \vdash x := \mathbf{new}\ C(e)}$$

Fig. 7. Program typing rules

4.2 Expression typing

We derive a typing judgement of the form $C \vdash e \,:\, \ell$, which means with assumptions in C, expression e has a label ℓ. The meaning of ℓ is given by its evaluation in a program state.

Figure 6 shows the typing rules. They are similar to those in the type system by Volpano et al., however, typing judgements are extended by the set C. Since we deal with labels instead of simple domains, we make use of the derived label order instead of the domain order from the lattice, as can be seen in the subtyping rule that is used to "adjust" the label of an expression to a higher label.

The label for $x.f$ depends on the domain of f assigned by the environment $\mathsf{fieldlevel}$. If $\mathsf{fieldlevel}(f) = this.\delta$, the substitution yields the label $x.\delta$. Since x is a reference, we have to pay special attention to indirect flows from the referer to the referee induced by access to memory locations. Therefore, we form the least upper bound with the domain of the reference x itself.

4.3 Program typing

The program typing rules are shown in figure 7. A typing judgement $C, k \vdash P$ says program P is secure for program states that satisfy C. The domain k is the *program counter label* [6]; it is a lower bound of the labels of the variables and locations assigned in P and is used to capture indirect flows induced by branching statements.

The rules SUB, SKIP, SEQ, ASSIGN, IF, and WHILE are the same as in the

Volpano-Smith-Irvine system. However, for assignments to a variable x, we need to ensure that $x.\delta$ does not appear in a label in C: if there is any assumption about $x.\delta$ in the constraint set, overwriting x might invalidate that assumption. If, on the other hand, $x.\delta \notin C$, then we cannot derive anything about $x.\delta$ other than tautologies that are satisfied in every program state, hence overwriting x is safe.

The same argument holds for the NEW rule. Moreover, the argument e must be L, since that is the domain of the δ field it is assigned to.

The rule for label tests IF-LABEL is a specialised form of the standard IF rule. The only difference is that the constraint set C is extended by the assumption (ℓ_1, ℓ_2) for subprogram P_1, as explained above.

The labels from the expression typing rules actually only surface in field assignment rule PUT-FIELD. The label of the field is determined as in the expression typing rule for $x.f$. The assigned expression e must be typable with that label. Since write access to a field also indirectly reveals information about the reference variable x itself, the rule requires that the domain of x is dominated by the label of the field. The program counter k must be a domain and a lower bound of the label of f. Finally, the rule prohibits assignments to the δ field to prevent the relabeling of high security data as public. Also, a change of $x.\delta$ could render assumptions about it in C invalid.

4.4 Soundness

As mentioned above, C stores abstract information about domain variables in a state. If $C, k \vdash P$ can be derived such that $(\ell_1, \ell_2) \in C$, the execution of program P is secure in states σ where $[\![\ell_1]\!]_\sigma \sqsubseteq [\![\ell_2]\!]_\sigma$ holds. Hence we only need to consider states that satisfy the assumptions in C. The proof of the following theorems can be found in appendix A.

An expression typing judgement $C \vdash e : \ell$ means that the expression e only depends on information of label ℓ or below. If the evaluation of ℓ is L in two equivalent states, e only depends on L data, hence its value must be indistinguishable between both states. Unfortunately, the evaluation of ℓ may differ even between equivalent states. Hence we restrict the usual indistinguishability property to states in which the evaluation of ℓ is equal.

Theorem 4.3 (Soundness of expression typing) *If $C \vdash e : \ell$, then for all states σ and σ' that satisfy C and all partial bijections β such that $\sigma \sim_\beta \sigma'$, $[\![\ell]\!]_\sigma = [\![\ell]\!]_{\sigma'} = L$ implies $[\![e]\!]_\sigma \sim_\beta [\![e]\!]_{\sigma'}$.*

The program typing judgement $C, k \vdash P$ finally expresses that the program P is noninterfering if the initial states satisfy the constraint set C. If $k = H$, it additionally ensures P does not change low-security variables or fields. Both properties are formalised in the final two theorems, which can be proven by induction over the operational semantics.

Theorem 4.4 (Invisibility of high security changes) *If $C, H \vdash P$, then for all states σ_1, σ_2 such that σ_1 satisfies C and $\sigma_1 \xrightarrow{P} \sigma_2$, it holds $\sigma_1 \sim_{\mathrm{id}} \sigma_2$, where id is the identity function restricted to $\mathrm{dom}(h_1)$.*

Theorem 4.5 (Soundness of program typing) *If $C, k \vdash P$, then for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ such that σ_1 and σ_1' satisfy C and for all partial bijections β,*

$$\sigma_1 \sim_\beta \sigma_1' \;\wedge\; \sigma_1 \xrightarrow{P} \sigma_2 \;\wedge\; \sigma_1' \xrightarrow{P} \sigma_2' \;\Rightarrow\; \sigma_2 \sim_\gamma \sigma_2'$$

for some partial bijection $\gamma \supseteq \beta$.

Note that from $C, k \vdash P$, we can always derive $C, L \vdash P$ by the subsumption rule. If additionally $C = \emptyset$, the satisfiability requirement for the program states holds trivially, hence the noninterference property from section 3 holds.

Corollary 4.6 *If $\emptyset, L \vdash P$, then P is noninterferent.*

5 Compilation to bytecode

Our language is mainly an extension of the high-level language presented by Barthe et al. [4], hence the JVM-like instruction set and the compilation function they present can be reused. A bytecode program may manipulate values on a stack, load and store them into variables, and jump to other program instructions.

The authors translate the type system to the bytecode level by determining the control dependence regions and junction points of a compiled program, which specify which bytecode sequences represent a compiled subprogram of a conditional statement. This can be used to define a security environment $se : \mathcal{PP} \to \mathcal{D}$ that associates every instruction with a level that represents the program counter label of the high-level statement to which the instruction belongs.

The type system derives judgements of the form $se, i \vdash st \Rightarrow st'$, where st and st' are stack typings that assign a domain to each value on the stack. Given a security environment se, executing the instruction at program position i with given stack typing st results in a new stack with typing st'. The type system thus forms an abstract transition system. A program is secure if a stack typing can be assigned to each program point.

In the proof-carrying code setting, the type derivation can be delivered as a formal proof object to the code consumer. For the type system above, a bytecode analyser and verifier as well as a soundness proof formalised in Coq have been presented by Barthe, Pichardie, and Rezk [2] as part of the Mobius project [12].

Our ongoing work includes an extension of that bytecode type system: in a similar fashion to se, we translate the constraint set C to a function $C : \mathcal{PP} \to \mathcal{L} \times \mathcal{L}$ that tells for each instruction which label order assumptions hold at that point. The problem is that there is no primitive bytecode instruction for label tests like $\ell_1 \sqsubseteq \ell_2$; instead, those tests are decomposed to a number of label operation instructions. Our type system only recognises label check instruction sequences that have been generated by the compiler. Alternatively, the compiler could annotate the bytecode with information about the labels that are created.

The `getfield` f instruction of the bytecode language accesses the field f of any object that is on top of the stack. The extended type system with dynamic domains, however, needs to know from which variable the object was loaded in order to determine the correct label for f. Again, this can be done by giving typing rules

for larger code fragments.

We have already extended the language to arbitrary field access with fully qualified names of the form $x.f_1.f_2.\ldots.f$. Labels may take the extended form $x.f_1.f_2.\ldots.\delta$. This requires an extension of the field access rules both in the high-level and the bytecode type system. Also, the soundness proof is more complicated. Special attention has to be paid to field assignments: a field may not be overwritten if it is part of a qualified name of a domain variable in the constraint set.

6 Discussion

Several challenges arise for information flow security in the presence of objects whose security domains are not known before runtime. In this paper, we presented a language and security notion that allows for writing programs that safely access dynamically typed objects.

This section outlines further benefits of the feature in mobile code scenarios, and describe further steps that need to be taken to embed dynamic domains into proof carrying code techniques.

Dynamic security policies

As mentioned in the introduction, the language allows for programs that are secure for varying security policies. Assume the set $\mathcal{D}$ consists of additional domains $k_1, k_2, \ldots$ such that nothing is statically known about their order except for $L \sqsubseteq k_i$ and $k_i \sqsubseteq H$. Since the type system does not rely on a given flow relation $\sqsubseteq$, any well-typed program guards every flow that involves these additional domains, and is therefore secure for any given security policy.

Dynamic object creation and selection

The introductory example showed a program that chooses a file using a function `openFile`. If we extend our language with arrays, we can model the function as an array `open` that maps every file name to the memory location of the heap object that models the respective file. Again, special care has to be taken with respect to indirect flows: when a high security file name is computed, looking it up in the array should return a high security memory location, so that write access to low security fields of that object (which would reveal information about the file name) is prohibited by the type system.

The creation of a file can be modelled with the **new File**(ℓ) construct, where ℓ is the label of the newly created file. The returned location can then be bound to the array element `open[fn]` to simulate the creation of a file with filename fn.

Methods

To move towards a full object-oriented language, we need to introduce methods. For this, an environment μ maps method identifiers to their operational semantics $(s, h) \to (h', v)$. The initial state s is defined on the local variables given as method arguments as well as on the special variable *this* for the object itself, and v is the return value of the method.

Methods come equipped with a type $\overrightarrow{k_1} \xrightarrow{C,\, k_2} k_3$, where $\overrightarrow{k_1}$ contains the domains of each argument variable and *this*, C contains the constraints for which the method is secure, k_2 is the lower bound of the side effects and k_3 is the return value domain.

Details of type checking method calls in general are discussed e.g. in [3]; here, we focus on the constraint set. When a method is called, the labels in the caller's constraint set need to be transformed, since they usually refer to variable names that are not valid in the method's body. For example, if a variable x is passed as an argument that is known as local variable y within the method, we can translate a label $x.\delta$ to the label $y.\delta$. Other types of arguments possibly require a more sophisticated constraint set translation. In any case, the translated constraint set has to subsume the constraint set given by the method's type.

Similarly, we can built upon existing work on type-preserving compilation when including additional language features like exceptions. Another idea is to extend our language in the direction of λ_{DSec}, determining which higher-order features can be translated into our language and to the bytecode level.

Improving the analysis

The label order inferences of the type system can be improved by storing aliasing information. For example, if x and y are references, after the execution of $x := y$ or $x := \mathbf{new}\ C(y.\delta)$, we know that $x.\delta = y.\delta$. This requires an extension of the type system that tracks the pre- and post-sets of constraints for a given program.

Secure software specification

Another topic is to explore how the dynamic domains can be abstracted and included in a software design specification process that takes flow security considerations into account. A security framework, which includes the specification of information flow secure software and a language-based security analysis, are goals of the InfoZert project [5]. Our aim is to integrate this work into the framework.

Acknowledgement

References

[1] Banerjee, A. and D. Naumann, *Stack-based access control and secure information flow* (2003).

[2] Barthe, G., D. Pichardie and T. Rezk, *A certified lightweight non-interference java bytecode verifier*, in: R. D. Nicola, editor, *ESOP*, Lecture Notes in Computer Science **4421** (2007), pp. 125–140.

[3] Barthe, G. and T. Rezk, *Non-interference for a jvm-like language* (2005), pp. 103–112.

[4] Barthe, G., T. Rezk and D. A. Naumann, *Deriving an information flow checker and certifying compiler for java* (2006), pp. 230–242.

[5] Beringer, L., R. Grabowski, M. Hofmann, A. Knapp, F. Lasinger and D. von Oheimb, *The InfoZert project*, http://www.pst.ifi.lmu.de/Research/current-projects/infozert.

[6] Denning, D. E., "Cryptography and Data Security," Addison-Wesley, 1982.

[7] Denning, D. E. and P. J. Denning, *Certification of programs for secure information flow*, Commun. ACM **20** (1977), pp. 504–513.

[8] Heintze, N. and J. G. Riecke, *The SLam calculus: programming with secrecy and integrity*, in: ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998* (1998), pp. 365–377.

[9] Myers, A. C., *Jflow: Practical mostly-static information flow control.*, in: *POPL*, 1999, pp. 228–241.

[10] Necula, G. C., *Proof-carrying code*, in: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL 1997)* (1997), pp. 106–119.

[11] Sabelfeld, A. and A. C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications – special issue on Formal Methods for Security **21** (2003), pp. 5 – 19.

[12] The Mobius project, *Mobility, Ubiquity and Security*, http://mobius.inria.fr/.

[13] Volpano, D., G. Smith and C. Irvine, *A sound type system for secure flow analysis*, J. Computer Security **4** (1996), pp. 167–187.

[14] Zheng, L. and A. C. Myers, *Dynamic security labels and static information flow control*, Int. J. Inf. Secur. **6** (2007), pp. 67–84.

A   Soundness proofs

Theorem 4.3 (repeated). If $C \vdash e : \ell$, then for all states σ and σ' that satisfy C and all partial bijections β such that $\sigma \sim_\beta \sigma'$, $[\![\ell]\!]_\sigma = [\![\ell]\!]_{\sigma'} = L$ implies $[\![e]\!]_\sigma \sim_\beta [\![e]\!]_{\sigma'}$.

Proof. By induction over the derivation of the typing judgement.

Let $\sigma = (s, h)$ and $\sigma' = (s', h')$ be two states that satisfy C.

- $C \vdash v : L$: Since $[\![v]\!]_\sigma = [\![v]\!]_{\sigma'} = v$, $[\![v]\!]_\sigma \sim_\beta [\![v]\!]_{\sigma'}$ holds trivially

- $C \vdash x : \Gamma(x)$: If $\Gamma(x) = [\![\Gamma(x)]\!]_\sigma = [\![\Gamma(x)]\!]_{\sigma'} = L$, then $[\![x]\!]_\sigma \sim_\beta [\![x]\!]_{\sigma'}$ follows from $\sigma \sim_\beta \sigma'$.

- $C \vdash x.f : \mathsf{fieldlevel}(f)[x/this] \sqcup \Gamma(x)$: Let $\ell' = \mathsf{fieldlevel}(f)[x/this]$.
  If $[\![\ell' \sqcup \Gamma(x)]\!]_\sigma = [\![\ell' \sqcup \Gamma(x)]\!]_{\sigma'} = L$, then $\Gamma(x) = L$ and $[\![\ell']\!]_\sigma = [\![\ell']\!]_{\sigma'} = L$. By state equality, we get $[\![x.f]\!]_\sigma = h(s(x))(f) \sim_\beta h'(s'(x))(f) = [\![x.f]\!]_{\sigma'}$.

- $C \vdash e_1 \circ e_2 : \ell$: Rule inversion gives use $C \vdash e_1 : \ell$ and $C \vdash e_2 : \ell$. By induction, we have $[\![e_1]\!]_\sigma \sim_\beta [\![e_1]\!]_{\sigma'}$ and $[\![e_2]\!]_\sigma \sim_\beta [\![e_2]\!]_{\sigma'}$, hence $[\![e_1 \circ e_2]\!]_\sigma \sim_\beta [\![e_1 \circ e_2]\!]_{\sigma'}$ by the definition of the binary operators.

- $C \vdash e : \ell'$ as derived by the subsumption rule: Rule inversion gives us $C \vdash \ell \leq \ell'$ and $C \vdash e : \ell$. With label order soundness, we have $[\![\ell]\!]_\sigma \sqsubseteq [\![\ell']\!]_\sigma = L$, hence $[\![\ell]\!]_\sigma = L$; the same holds for $[\![\ell]\!]_{\sigma'}$. By induction, we get $[\![e]\!]_\sigma \sim_\beta [\![e]\!]_{\sigma'}$.

$\square$

For the soundness proof of the program typing rules, we need an auxiliary lemma.

Lemma A.1 *If $C, k \vdash P$, then for all states σ_1, σ_2 such that $\sigma_1 \xrightarrow{P} \sigma_2$, if σ_1 satisfies C, then σ_2 satisfies C.*

Proof. By induction over the operational semantics. $\square$

Theorem 4.4 (repeated). If $C, H \vdash P$, then for all states σ_1, σ_2 such that σ_1 satisfies C and $\sigma_1 \xrightarrow{P} \sigma_2$, it holds $\sigma_1 \sim_{\mathsf{id}} \sigma_2$, where id is the identity function restricted to $\mathrm{dom}(h_1)$.

Proof. By induction over the derivation of the operational semantics. Without loss of generality, we assume $C, H \vdash P$ was not derived by the subsumption rule.

- $P = \mathbf{skip}$. We have to show $\sigma_1 \sim_{\mathrm{id}} \sigma_1$, which follows from the definition of state equivalence.

- $P = P_1 \; ; \; P_2$. Then we have $\sigma_1 \xrightarrow{P_1} \sigma_2$ and $\sigma_2 \xrightarrow{P_2} \sigma_3$ from the operational semantics and $C, H \vdash P_1$ and $C, H \vdash P_2$ from the type rules. By induction and together with lemma A.1, we get $\sigma_1 \sim_{\mathrm{id}} \sigma_2$ and $\sigma_2 \sim_{\mathrm{id}} \sigma_3$. Transitivity of equivalence results in $\sigma_1 \sim_{\mathrm{id}} \sigma_3$.

- $P = x := e$. We get $s_2 = s_1[x \mapsto \llbracket e \rrbracket_{s_1,h_1}]$ and $\Gamma(x) = H$. By the definition of state equivalence, we have $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.

- $P = \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$.
  - Case $\llbracket e \rrbracket_{\sigma_1} = 0$: $\sigma_1 \xrightarrow{P_1} \sigma_2$ and $C, H \vdash P_1$ imply by induction $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.
  - Case $\llbracket e \rrbracket_{\sigma_1} \neq 0$: $\sigma_1 \xrightarrow{P_2} \sigma_2$ and $C, H \vdash P_2$ imply by induction $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.

- $P = \mathbf{while}\ e\ \mathbf{do}\ P$.
  - Case $\llbracket e \rrbracket_{\sigma_1} = 0$: $\sigma_1 \sim_{\mathrm{id}} \sigma_1$ holds trivially.
  - Case $\llbracket e \rrbracket_{\sigma_1} \neq 0$: We have $\sigma_1 \xrightarrow{P} \sigma_2$ and $C, H \vdash P$, hence by induction $\sigma_1 \sim_{\mathrm{id}} \sigma_2$. Also, we have $\sigma_2 \xrightarrow{\mathbf{while}\ e\ \mathbf{do}\ P} \sigma_3$ and the original $C, H \vdash \mathbf{while}\ e\ \mathbf{do}\ P$. Again by induction and with lemma A.1, $\sigma_2 \sim_{\mathrm{id}} \sigma_3$. Transitivity gives $\sigma_1 \sim_{\mathrm{id}} \sigma_3$.

- $P = \mathbf{if}\ \ell_1 \sqsubseteq \ell_2\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$.
  - Case $\llbracket \ell_1 \rrbracket_{\sigma_1} \sqsubseteq \llbracket \ell_2 \rrbracket_{\sigma_1}$: Then σ_1 satisfies $C \cup \{(\ell_1, \ell_2)\}$. From $\sigma_1 \xrightarrow{P_1} \sigma_2$, we get by induction $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.
  - Case $\llbracket \ell_1 \rrbracket_{\sigma_1} \not\sqsubseteq \llbracket \ell_2 \rrbracket_{\sigma_1}$: From $\sigma_1 \xrightarrow{P_2} \sigma_2$, we get by induction $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.

- $P = x.f := e$. Let $\ell_f = \mathsf{fieldlevel}(f)[x/this]$. Since $C \vdash H \leq \ell$, we have $\llbracket \ell_f \rrbracket_\sigma = \llbracket \ell_f \rrbracket_{\sigma'} = H$. By the definition of state equivalence, it follows $\sigma_1 \sim_{\mathrm{id}} \sigma_2$.

- $P = x := \mathbf{new}\ C(e)$. For the variable x, it holds the same argument as for the ordinary assign rule. For the heap, $h_1 \sim_{\mathrm{id}} h_2$ as the heap has not been changed for all $m \in \mathrm{dom}(h_1)$.

$\square$

Theorem 4.5 (repeated). If $C, k \vdash P$, then for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ such that σ_1 and σ_1' satisfy C and for all partial bijections β,

$$\sigma_1 \sim_\beta \sigma_1' \ \wedge\ \sigma_1 \xrightarrow{P} \sigma_2 \ \wedge\ \sigma_1' \xrightarrow{P} \sigma_2' \ \Rightarrow\ \sigma_2 \sim_\gamma \sigma_2'$$

for some partial bijection $\gamma \supseteq \beta$.

Proof. By induction over the derivation of the operational semantics. Without loss of generality, we can assume $C, k \vdash P$ was not derived by the subsumption rule.

If $k = H$, theorem 4.4 gives $\sigma_1 \sim_{\mathrm{id}} \sigma_2$ and $\sigma_1' \sim_{\mathrm{id}} \sigma_2'$. With definiton of state equivalence, we get $\sigma_2 \sim_\beta \sigma_2'$. In the following, we therefore assume $k = L$.

- $P = \textbf{skip}$. From the assumption follows $\sigma_1 \sim_\beta \sigma_1'$.

- $P = P_1 \,;\, P_2$. Then we have $\sigma_1 \xrightarrow{P_1} \sigma_2$ and $\sigma_2 \xrightarrow{P_2} \sigma_3$ as well as $\sigma_1' \xrightarrow{P_1} \sigma_2'$ and $\sigma_2' \xrightarrow{P_2} \sigma_3'$ from the operational semantics and $C, k \vdash P_1$ and $C, k \vdash P_2$ from the type rules. By induction, we get $\sigma_2 \sim_\beta \sigma_2'$. From lemma A.1, we know σ_2 and σ_2' satisfy C. Again by induction, we get $\sigma_3 \sim_\beta \sigma_3'$.

- $P = x := e$. We get $s_2 = s_1[x \mapsto \llbracket e \rrbracket_{s_1, h_1}]$ and $s_2' = s_1'[x \mapsto \llbracket e \rrbracket_{s_1', h_1'}]$. From the typing rules, we get $\Gamma(x) = L$ and $C \vdash e \,:\, k$. With theorem 4.3, we get get $\llbracket e \rrbracket_{s_1, h_1} = \llbracket e \rrbracket_{s_1', h_1'}$. Therefore, $s_2(x) = s_2'(x)$, therefore $(s_2, h_1) \sim_\beta (s_2', h_1')$.

- $P = \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2$. We get $C \vdash e \,:\, L$. We get with theorem 4.3 $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'}$.
  - Case $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'} = 0$: $\sigma_1 \xrightarrow{P_1} \sigma_2$ and $\sigma_1' \xrightarrow{P_1} \sigma_2'$ and $C, L \vdash P_1$ imply by induction $\sigma_2 \sim_\beta \sigma_2'$.
  - Case $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'} \neq 0$: $\sigma_1 \xrightarrow{P_2} \sigma_2$ and $\sigma_1' \xrightarrow{P_2} \sigma_2'$ and $C, L \vdash P_2$ imply by induction $\sigma_2 \sim_\beta \sigma_2'$.

- $P = \textbf{while } e \textbf{ do } P$. We get $C \vdash e \,:\, L$, and with theorem 4.3 $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'}$.
  - Case $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'} = 0$: We have to show $\sigma_1 \sim_\beta \sigma_1'$, which follows from the assumption.
  - Case $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_1'} \neq 0$: We have $\sigma_1 \xrightarrow{P} \sigma_2$ and $\sigma_1' \xrightarrow{P} \sigma_2'$ and $C, L \vdash P$, hence by induction $\sigma_2 \sim_\beta \sigma_2'$. Also, we have $\sigma_2 \xrightarrow{\textbf{while } e \textbf{ do } P} \sigma_3$ and $\sigma_2' \xrightarrow{\textbf{while } e \textbf{ do } P} \sigma_3'$ and the original $C, L \vdash \textbf{while } e \textbf{ do } P$. Also, we know from lemma A.1 that σ_2 and σ_2' satisfy C. By induction, $\sigma_3 \sim_\beta \sigma_3'$.

- $P = \textbf{if } \ell_1 \sqsubseteq \ell_2 \textbf{ then } P_1 \textbf{ else } P_2$. We have $C \cup \{(\ell_1, \ell_2)\}, L \vdash P_1$ and $C, L \vdash P_2$. With $C \vdash \ell_1 \,:\, L$ and $C \vdash \ell_2 \,:\, L$, we get from theorem 4.3 and with operational semantics that $\llbracket \ell_1 \sqsubseteq \ell_2 \rrbracket_{\sigma_1} = \llbracket \ell_1 \sqsubseteq \ell_2 \rrbracket_{\sigma_1'}$. Let the evaluation of the test be b.
  - If b is true: Then σ_1 and σ_1' satisfy $C \cup \{(\ell_1, \ell_2)\}$. From $\sigma_1 \xrightarrow{P_1} \sigma_2$ and $\sigma_1' \xrightarrow{P_1} \sigma_2'$, we get by induction $\sigma_2 \sim_\beta \sigma_2'$.
  - If b is false: From $\sigma_1 \xrightarrow{P_2} \sigma_2$ and $\sigma_1' \xrightarrow{P_2} \sigma_2'$, we get by induction $\sigma_2 \sim_\beta \sigma_2'$.

- $P = x.f := e$. If $\ell = \mathsf{fieldlevel}(f)[x/\mathit{this}]$ evaluates to H in both states, then the field update does not affect state equality.

  Let ℓ evaluate to L in one state. Then it must also evaluate to L in the other state: $\mathsf{fieldlevel}(f)[x/\mathit{this}]$ can either be a constant L, or $x.\delta$. $C \vdash \Gamma(x) \leq \ell$ implies that $\Gamma(x) = L$, hence $x.\delta$ evaluates to the same domain (L) in both states.

  With theorem 4.3, we know $\llbracket e \rrbracket_{\sigma_1} \sim_\beta \llbracket e \rrbracket_{\sigma_1'}$, and hence $\sigma_2 \sim_\beta \sigma_2'$.

- $P = x := \textbf{new } C(e)$. With theorem 4.3, we know $\llbracket e \rrbracket_{\sigma_1} \sim_\beta \llbracket e \rrbracket_{\sigma_1'}$, hence for the newly create objects o and o', we have $o \sim_\beta o'$. Let m and m' be fresh in h_1 and h_1'. Then there exists a partial bijection $\gamma = \beta \cup \{(m, m')\} \supseteq \beta$ such that $h_2 \sim_\gamma h_2'$. Since we have $s_2(x) = m \sim_\gamma m' = s_2'(x)$, we get $\sigma_2 \sim_\gamma \sigma_2'$.

$\square$