Preventing timing leaks through transactional branching instructions

Gilles Barthe [1]

INRIA Sophia-Antipolis, Project EVEREST,
2004, Route de Lucioles, BP 93, Sophia-Antipolis Cedex, France

Tamara Rezk [2]

INRIA Sophia-Antipolis, Project EVEREST,
2004, Route de Lucioles, BP 93, Sophia-Antipolis Cedex, France

Martijn Warnier [3]

Radboud University Nijmegen, SoS group,
Toernooiveld 1, 6500 GL, Nijmegen, The Netherlands

Abstract

Timing channels constitute one form of covert channels through which programs may be leaking information about the confidential data they manipulate. Such timing channels are typically eliminated by design, employing ad-hoc techniques to avoid information leaks through execution time, or by program transformation techniques, that transform programs that satisfy some form of non-interference property into programs that are time-sensitive termination-sensitive non-interfering. However, existing program transformations are thus far confined to simple languages without objects nor exceptions.

We introduce a program transformation that uses transaction mechanisms to prevent timing leaks in sequential object-oriented programs. Under some strong but reasonable hypotheses, the transformation preserves the semantics of programs and yields for every termination-sensitive non-interfering program a time-sensitive termination-sensitive non-interfering program.

Key words: Non-interference, Timing leaks, Security, Program transformation, Semantics

[1] Email: Gilles.Barthe@sophia.inria.fr
[2] Email: Tamara.Rezk@sophia.inria.fr
[3] Email: warnier@cs.ru.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Modern programming languages for mobile code often rely on type systems to enforce basic safety policies; on the other hand, basic security properties such as confidentiality, integrity and availability are notoriously hard to enforce, and only a few languages attempt to offer static enforcement mechanisms for these properties. Thus it is important to develop tools that help enforcing properties that relate to the aforementioned security properties.

This paper is concerned with non-interference [9], a high-level property that guarantees confidentiality of programs by distinguishing between low (public) and high (secret) program variables, and by requiring that all high security level variables are completely independent of low variables. In other words, a program is non-interfering if it is impossible to learn the value of high level variables by observing low level variables.

Many methods proposed in the literature (see Sabelfeld and Myers [17] for an overview) deal with *termination insensitive* non-interference. This is a weak form of non-interference that only considers normal termination modes. A program is deemed termination insensitive non-interfering if all high variables are independent of low variables, provided the program terminates normally. In other words, the non-interference property does not consider those cases where a program hangs or terminates abruptly (e.g. via an exception).

A somewhat stronger version of non-interference does take the termination behavior of programs into account resulting in the notion of *termination sensitive* non-interference. Thus in this case non-interference ensures that no matter how a program terminates (or hangs) high variables are independent of low ones.

Still stronger versions of non-interference also consider *covert channels* [13]. If a program leaks secret information via channels that are not intended for communication we speak of information leakage via covert channels. Both resource consumption (memory/CPU) and timing behavior of a program can be used as a covert channel.

This paper deals with termination-sensitive non-interference and one covert channel: timing behavior. This form of non-interference is called *time-sensitive termination-sensitive non-interference*, and ensures termination-sensitive non-interference and the absence of timing channels. Such timing channels can be used to leak sensitive information: for example, Kocher [11] showed that certain implementations of encryption algorithms can leak information about the used key via timing behavior. Unfortunately, timing leaks are hard to avoid by design, in particular because even the slightest difference in execution time can be made observable by putting it inside a loop, thereby potentially leaking secret information. They are also very hard to detect, and static enforcement mechanisms for non-interference do not consider timing channels. In fact, the main trend in avoiding timing channels is to use a program transformation that transforms termination-insensitive non-interfering programs into time-

sensitive termination-sensitive non-interfering programs. However, existing results are limited to programs that only exploit a limited set of features.

The main result of the paper is a program transformation method that eliminates timing leaks in sequential object oriented languages with exceptions. Our method for enforcing non-interference is based on (nested) transaction mechanisms [15]. The basic idea is to transform conditionals that depend on high expressions, i.e. expressions that depend on high variables, into conditionals in which each branch performs two transactions (one transaction for each branch in the original conditional statement), one of which is committed, namely the one that would have been executed in the original statement. Formally, the idea is to transform a branching statement of the form

$$\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$$

into the statement

$$\begin{aligned}\mathsf{if}\ e\ \ &\mathsf{then}\ \mathsf{beginT};\ c_2'\ \mathsf{abortT};\mathsf{beginT};\ c_1';\ \mathsf{commitT}\\ &\mathsf{else}\ \mathsf{beginT};\ c_1';\ \mathsf{abortT};\mathsf{beginT};\ c_2'\ ;\mathsf{commitT}\end{aligned}$$

where beginT starts a new transaction and abortT and $\mathsf{commitT}$ respectively aborts and commits a transaction, and c_1' and c_2' are respectively the statements c_1 and c_2 transformed in the same manner.

The proposed transformation offers several advantages. First of all, the transformation is correct in the sense that termination-insensitive non-interfering programs are transformed into time-sensitive termination-sensitive non-interfering programs.

Second of all, the method is applicable to sequential object-oriented languages with exceptions and method calls, and therefore handles a fragment of the language that is significantly more expressive than the languages considered in previous works, see below. In particular, this is the first work considering dynamic object creation. Furthermore, the transformation is applicable to structured languages and intermediate or low level languages (but for the sake of clarity, we choose to present the transformation at source code level).

Third of all, the transformation is independent of the technique used to enforce termination-insensitive non-interference. More specifically, the transformation does not rely on the fact that programs are verified with an information flow type system as in Volpano and Smith's work [18], or using a program logic as in Barthe, D'Argenio and Rezk's approach [6].

On a more negative side our translation raises several questions, which are discussed in Section 5.

Related work

Agat [1,3] suggests an approach to remove timing leaks based on program transformation. In essence his approach involves dummy assignments and branching instructions which take exactly the same time as normal assignments and branching instructions. By padding a program with these dummy instructions he can prove that the resulting program will always execute in a time which only depends on non-secret variables, thereby removing all timing leaks. An additional type system then enforces that well typed padded programs are time-sensitive termination-sensitive non-interfering, under the restriction that all guards of while commands are typed minimal. The programming language Agat considers is an imperative language with arrays, but without objects or exceptions. In [2] Agat also implemented his approach, using (part of) Java byte code as his programming language.

Recently, Hedin and Sands [10] have extended Agat's work towards an object oriented language. However, for now, exceptions are not supported.

Köpf and Mantel [12] exposed ideas of how to improve type systems to eliminate timing leaks by incorporating unification. They look at a simple imperative language without objects.

Contents

The remainder of this paper is organized as follows: Section 2 introduces a simple imperative programming, Section 3 shows how to transform programs in such a way that timing-leaks are prevented. Section 4 describes how to extend this translation to an object oriented language with exceptions. Section 5 gives some general observations about our approach and ideas how to implement the translation for (sequential) Java. We end with conclusions and future work.

2 Language

We first consider a sequential imperative language whose set Expr of *expressions* and Comm of *commands* are given by the syntaxes in Figure 1. In this language, a program P is a declaration of the form $P(\vec{x}) := c$. The operational semantics of programs is given by a small step operational semantics that captures one step execution of the program, and relate states, execution time and results. In our setting, results are simply values, and we let Res denote the set of results. The set State of states is defined as the set of pairs of the form $\langle c, \rho\rangle$ where c is in Comm, ρ is a mapping from local variables from a set of variables $\mathcal{X}$ to values. We distinguish a special variable res to store results of execution of programs, as is defined below. Finally, the execution time of the program is modeled using a commutative monoid $(T, +, 0)$.

Formally, the operational semantics is defined through a relation $s \rightsquigarrow_t r$, where $s \in \mathsf{State}$, $r \in \mathsf{Res} \cup \mathsf{State}$ and $t \in T$, with intuitive meaning that s

$$e ::= x \mid n \mid e_1 \; op \; e_2 \mid$$

$$c ::= x := e \mid c_1; c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{return}\ e \mid \epsilon$$

where op is either a primitive operation $+, \times$, a comparison operation $<, \leq, =$, or the (unconditional) boolean connectives | and & and ϵ is the empty program (skip).

Fig. 1. The language

evaluates to r in time t. The closure $\rightsquigarrow_t^*$ is then defined inductively by the clauses:

- if $s \rightsquigarrow_t s'$ then $s \rightsquigarrow_t^* s'$;
- if $s \rightsquigarrow_t^* s'$ and $s' \rightsquigarrow_{t'}^* s''$ then $s \rightsquigarrow_{t+t'}^* s''$.

Finally, we define an evaluation relation $\Downarrow$ between states, results and execution time and set $\langle c, \rho\rangle \Downarrow_t v$ iff $\langle c, \rho\rangle \rightsquigarrow_t^* \langle \epsilon, \rho'\rangle$ with $\rho'(res) = v$ and $v \in \mathsf{Res}$. In the sequel, we often write $P, \rho \Downarrow_t r$ instead of $\langle c, \rho\rangle \Downarrow_t r$ where $P(\vec{x}) = c$. Further, we simply write $P, \rho \Downarrow_t$ when the result of the evaluation is irrelevant, i.e. as a shorthand for $\exists r.\ P, \rho \Downarrow_t r$. Finally, for every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f \oplus \{x \mapsto y\}$ denote the unique function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$.

The rules of the operational semantics are given in Figure 2. The rules are standard, except for execution time for which there are several possible models. In our model, each command has its own execution time; for example, the execution time of $x := e$ is equal to the sum of the execution time of e and of some constant $t_{:=}$. More refined models allow the execution time of each instruction to be parametrized by the state, or even by execution history; for example, in the above example $t_{:=}$ would become a function. It is possible to extend our results to such execution models, by imposing suitable equational constraints on these functions.

3 Transforming out timing leaks

Our method for transforming out timing leaks is based on (nested) transaction mechanisms [15]. Transactions have the ACID property (Atomicity, Coherence, Isolation and Durability) which makes them ideal for use in an approach for enforcing (stronger forms of) non-interference.

3.1 Transactions

Transactions allow a programmer to view code blocks as atomic and perform all updates inside such a transaction block as conditional. Only after an explicit *commit* command an update is really carried out. If the programmer

$$\frac{}{\langle x, \rho\rangle \rightsquigarrow_{t_{R_1}} \langle \rho(x), \rho\rangle}$$

$$\frac{v \ op \ v' = v''}{\langle v \ op \ v', \rho\rangle \rightsquigarrow_{t_{OP}} \langle v'', \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \rightsquigarrow_t \langle e_1', \rho\rangle}{\langle e_1 \ op \ e_2, \rho\rangle \rightsquigarrow_t \langle e_1' \ op \ e_2, \rho\rangle}$$

$$\frac{\langle e_2, \rho\rangle \rightsquigarrow_t \langle e_2', \rho\rangle}{\langle v \ op \ e_2, \rho\rangle \rightsquigarrow_t \langle v \ op \ e_2', \rho\rangle}$$

$$\frac{}{\langle x := v, \rho\rangle \rightsquigarrow_{t_:=} \langle \epsilon, \rho \oplus \{x \mapsto v\}\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle e', \rho\rangle}{\langle x := e, \rho\rangle \rightsquigarrow_t \langle x := e', \rho\rangle}$$

$$\frac{\langle c_1, \rho\rangle \rightsquigarrow_t \langle c_1', \rho'\rangle}{\langle c_1; c_2, \rho\rangle \rightsquigarrow_t \langle c_1'; c_2, \rho'\rangle}$$

$$\frac{}{\langle \epsilon; c_2, \rho\rangle \rightsquigarrow_{t_\epsilon} \langle c_2, \rho'\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle true, \rho\rangle}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, \rho\rangle \rightsquigarrow_t \langle c; \mathsf{while}\ e\ \mathsf{do}\ c, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle false, \rho\rangle}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, \rho\rangle \rightsquigarrow_t \langle \epsilon, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle true, \rho\rangle}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \rho\rangle \rightsquigarrow_t \langle c_1, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle false, \rho\rangle}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \rho\rangle \rightsquigarrow_t \langle c_2, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle e', \rho\rangle}{\langle \mathsf{return}\ e, \rho\rangle \rightsquigarrow_t \langle \mathsf{return}\ e', \rho\rangle}$$

$$\frac{}{\langle \mathsf{return}\ v, \rho\rangle \rightsquigarrow_{t_R} \langle \epsilon, \rho \oplus \{res \mapsto v\}\rangle}$$

Fig. 2. Small Step Operational Semantics for the core language

desires so it is also possible to perform a roll-back to the state before the beginning of the transaction block via an explicit *abort* command.

The operational semantics of transactions is given in Figure 3. Note that transactions can be nested arbitrarily deep.

3.2 Problem statement and hypotheses

Since we are interested in ensuring non-interference, we assume that all expressions, including program variables, are classified with security levels high

$$\frac{\langle c, \rho\rangle \Downarrow_{t'} \langle \epsilon, \rho'\rangle}{\langle \mathsf{beginT}; c; \mathsf{abortT}, \rho\rangle \rightsquigarrow_{t_{ba}+t'} \langle \epsilon, \rho\rangle}$$

$$\frac{\langle c, \rho\rangle \Downarrow_{t'} \langle \epsilon, \rho'\rangle}{\langle \mathsf{beginT}; c; \mathsf{commitT}, \rho\rangle \rightsquigarrow_{t_{bc}+t'} \langle \epsilon, \rho'\rangle}$$

Fig. 3. Operational Semantics for transaction mechanisms

(secret) or low (public). Formally, we assume given a function sl that maps expressions to security levels. Furthermore, we assume that memories that coincide on their low parts yield equal results for the evaluation of low expressions. Formally, we introduce an equality $\simeq$ on memories, and set $\rho \simeq \rho'$ to hold iff $\rho(x) = \rho(x')$ for all variables x such that $\mathsf{sl}(x) = L$. Then, we assume that for all expressions e and memories ρ and ρ' such that $\rho \simeq \rho'$, we have

$$< e, \rho > \Downarrow_t v \quad \wedge \quad < e, \rho' > \Downarrow_{t'} v' \quad \Rightarrow \quad v = v'$$

Timing leaks in a program might occur when the branches of a conditional statement take different execution times. In our language, branches in execution are caused either by if-then-else or while commands. Due to our execution model, timing leaks can only occur when the conditional statement makes its test on a high expression. We only address the case of if-then-else statements that branch over high expressions, and restrict ourselves to low-recursive programs, where a program P is low-recursive iff it does not contain a statement of the form $\mathsf{while}\ e\ \mathsf{do}\ c$ with $\mathsf{sl}(e) = H$. While the restriction to low-recursive programs is rather severe, it is already present in the works of Agat [1,3] and Volpano and Smith [18].

3.3 The transformation

In order to avoid timing leaks caused by if-then-else statements, we use transactions to execute both branches of the statement. Thanks to an appropriate use of committing and aborting transactions, the transformation is semantics preserving up to termination.

The transformation of if-then-else statements is given in Figure 4, and is defined relative to a mapping sl that gives the security level of an expression or statement. The 'big' if-then-else's in the translation –`IF`, `THEN` and `ELSE`– belongs to the translation and not to the programming language, and is introduced so that we only use transactions for `if-then-else` blocks with a high conditional. For all other cases, the transformation is defined by the obvious recursive clause.

The transformation is semantics preserving up to termination of the transformed program. Indeed, the transformation may introduce non-termination,

$\mathcal{T}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ =$
$\quad$ `IF` $\mathsf{sl}(e) = \mathsf{L}$ `THEN` $\mathsf{if}\ e\ \mathsf{then}\ \mathcal{T}(c_1)\ \mathsf{else}\ \mathcal{T}(c_2)$
$\quad$ `ELSE` $\mathsf{if}\ e\ \mathsf{then}$
$\qquad$ `beginT;`$\mathcal{T}(c_2)$`;abortT;beginT;`$\mathcal{T}(c_1)$`;commitT; else`
$\qquad$ `beginT;`$\mathcal{T}(c_1)$`;abortT;beginT;`$\mathcal{T}(c_2)$`;commitT;`

Fig. 4. Translation $\mathcal{T}$ for removing timing leaks

as illustrated by the program

$$\mathsf{if}\ x_H > 0\ \mathsf{then}\ (\mathsf{if}\ x_H < 0\ \mathsf{then}\ \mathit{loop}\ \mathsf{else}\ \epsilon)\ \mathsf{else}\ \epsilon$$

Nevertheless we have:

Lemma 3.1 *For every program P, memory ρ, results $r, r' \in \mathsf{Res}$, and times $t, t' \in T$ such that $P, \rho \Downarrow_t r$ and $\mathcal{T}(P), \rho \Downarrow_{t'} r'$, we have $r = r'$.*

Proof. We prove that for every command c, memory ρ, results $r, r' \in \mathsf{Res}$, and times $t, t' \in T$ such that $c, \rho \Downarrow_t r$ and $\mathcal{T}(c), \rho \Downarrow_{t'} r'$, we have $r = r'$.

The proof proceeds by induction on the structure of c. It is straightforward and omitted.

□

3.4 Application to non-interference

This section shows that our transformation maps low-recursive non-interfering programs into time-sensitive termination-sensitive non-interfering programs.

Before establishing these results, we review the definitions of non-interference; in the sequel $\Uparrow$ denotes non-termination.

Definition 3.2 [Non-interference]

(i) A program P is termination-insensitive non-interfering, written $\mathsf{TINI}(P)$, if for every $\rho, \rho' \in \mathcal{X} \to \mathcal{V}$, and $v, v' \in \mathcal{V}$, we have $\langle P, \rho\rangle \Downarrow_t v$ and $\rho \simeq \rho'$ imply $\langle P, \rho'\rangle \Downarrow_{t'} v'$ and $v = v'$ or $P, \rho' \Uparrow$.

(ii) A program P is termination-sensitive non-interfering, written $\mathsf{TSNI}(P)$, if for every $\rho, \rho' \in \mathcal{X} \to \mathcal{V}$, and $v, v' \in \mathcal{V}$, we have $\langle P, \rho\rangle \Downarrow_t v$ and $\rho \simeq \rho'$ imply $\langle P, \rho'\rangle \Downarrow_{t'} v'$, and $v = v'$.

(iii) A program P is time-sensitive termination-sensitive non-interfering, written $\mathsf{TSTSNI}(P)$, if for every $\rho, \rho' \in \mathcal{X} \to \mathcal{V}$, and and $v, v' \in \mathcal{V}$, we have $\langle P, \rho\rangle \Downarrow_t v$ and $\rho \simeq \rho'$ imply $\langle P, \rho'\rangle \Downarrow_t v'$, and $v = v'$.

The difference between termination-insensitive and termination-sensitive non-interference is that the former only compares execution traces that terminate while the latter requires that the termination of the program is uniform in the high part of the memory. Note that $\mathsf{TSTSNI}(P)$ implies $\mathsf{TSNI}(P)$

(time-sensitive termination-sensitive non-interference imposes moreover that the execution time is uniform in the high part of the memory).

Since the transformation is semantic preserving up to termination, we prove that our transformation preserves termination-insensitive non-interference.

Corollary 3.3

For every low-recursive program P, $\mathsf{TINI}(P)$ *implies* $\mathsf{TINI}(\mathcal{T}(P))$.

Proof. Straightforward by Lemma 3.1. □

The transformation eliminates from programs timing leaks due to high if-then-else statements, provided that the program is non-interfering and low-recursive.

Theorem 3.4 *For all low-recursive non-interfering programs P, and memories ρ and ρ' such that $\rho \simeq_L \rho'$, we have $\mathcal{T}(P), \rho \Downarrow_t$ iff $\mathcal{T}(P), \rho' \Downarrow_t$.*

Proof. We prove that for all low-recursive commands c, and memories ρ and ρ' such that $\rho \simeq_L \rho'$, we have $\langle \mathcal{T}(c), \rho\rangle \rightsquigarrow_t \langle \epsilon, \rho_1\rangle$ iff $\langle \mathcal{T}(c), \rho'\rangle \Downarrow_t \langle \epsilon, \rho'_1\rangle$. The proof proceeds by structural induction on commands.

Case $c \equiv \mathcal{T}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$

- Suppose $\mathsf{sl}(e) = \mathsf{L}$ then

$$c \equiv \mathsf{if}\ e\ \mathsf{then}\ \mathcal{T}(c_1)\ \mathsf{else}\ \mathcal{T}(c_2)$$

  and by IH we have that for some t, $\langle \mathcal{T}(c_1), \rho\rangle \rightsquigarrow_t \langle \epsilon, \rho_1\rangle$ iff $\langle \mathcal{T}(c_1), \rho'\rangle \rightsquigarrow_t \langle \epsilon, \rho'_1\rangle$ and the same holds for c_2. Since $sl(e) = \mathsf{L}$ and $\rho \simeq_L \rho'$, by operational semantics e evaluates to the same value in memory ρ and ρ' and we conclude.
- Suppose $\mathsf{sl}(e) \neq \mathsf{L}$ then c is

$$\begin{array}{l}\mathsf{if}\quad e\ \mathsf{then}\\ \qquad \texttt{beginT;}\mathcal{T}(c_2)\texttt{;abortT;beginT;}\mathcal{T}(c_1)\texttt{;commitT;}\\ \mathsf{else}\\ \qquad \texttt{beginT;}\mathcal{T}(c_1)\texttt{;abortT;beginT;}\mathcal{T}(c_2)\texttt{;commitT;}\end{array}$$

  We have to prove that

$$\langle \texttt{beginT;}\mathcal{T}(c_2)\texttt{;abortT;beginT;}\mathcal{T}(c_1)\texttt{;commitT;}, \rho\rangle \rightsquigarrow_t \langle \epsilon, \rho_1\rangle \quad \text{iff}$$
$$\langle \texttt{beginT;}\mathcal{T}(c_1)\texttt{;abortT;beginT;}\mathcal{T}(c_2)\texttt{;commitT;}, \rho'\rangle \rightsquigarrow_t \langle \epsilon, \rho_1\rangle$$

($\Rightarrow$) Suppose that

$$\langle \texttt{beginT;}\mathcal{T}(c_2)\texttt{;abortT;beginT;}\mathcal{T}(c_1)\texttt{;commitT;}, \rho\rangle \rightsquigarrow_t \langle \epsilon, \rho_1\rangle$$

where $\langle \mathcal{T}(c_2), \rho\rangle \rightsquigarrow_{t_2} \langle \epsilon, \rho_2\rangle$ and $\langle \mathcal{T}(c_1), \rho\rangle \rightsquigarrow_{t_1} \langle \epsilon, \rho_2'\rangle$ and $t = t_1 + t_2 + t_{ba} + t_{bc}$. By IH $\langle \mathcal{T}(c_2), \rho'\rangle \rightsquigarrow_{t_2} \langle \epsilon, \rho_3\rangle$ and $\langle \mathcal{T}(c_1), \rho'\rangle \rightsquigarrow_{t_1} \langle \epsilon, \rho_3'\rangle$ hence we are done.

($\Leftarrow$) Analogous to previous case.

Case $c \equiv \mathsf{while}\ e\ \mathsf{do}\ c_1$ Because the program is low-recursive, we know that $\mathsf{sl}(e) = \mathsf{L}$ and since $\rho \simeq_L \rho'$, by operational semantics e evaluates to the same value in memory ρ and ρ'. If e evaluates to false, we are done. Otherwise, by IH $\langle \mathcal{T}(c_1), \rho\rangle \rightsquigarrow_{t_1} \langle \epsilon, \rho_1\rangle$ iff $\langle \mathcal{T}(c_1), \rho'\rangle \rightsquigarrow_{t_1} \langle \epsilon, \rho_1'\rangle$ holds. Since the program is non-interfering, we have that $\rho_1 \simeq_L \rho_1'$ and we can conclude.

Assignment, return and skip commands are trivial. The case for $c_1; c_2$ is straightforward by IH.

□

As a corollary of this result we prove that our transformation also transforms termination-insensitive non-interfering programs into (time-sensitive) termination-sensitive non-interfering programs.

Corollary 3.5 *For all low-recursive non-interfering programs P such that* $\mathsf{TINI}(P)$*, we have* $\mathsf{TSTSNI}(\mathcal{T}(P))$*.*

Proof. By Corollary 3.3, $\mathsf{TINI}(\mathcal{T}(P))$. By Theorem 3.4, for all low-recursive non-interfering programs P, and memories ρ and ρ' such that $\rho \simeq_L \rho'$, we have $\mathcal{T}(P), \rho \Downarrow_t$ iff $\mathcal{T}(P), \rho' \Downarrow_t$. □

3.5 Enforcing termination-sensitive non-interference

Our transformation yields time-sensitive termination-sensitive non-interfering programs provided the input programs are termination-insensitive non-interfering programs and low-recursive. In this section, we examine how the latter properties may be enforced on programs.

Volpano and Smith [18] provide a sound information flow type system for a simple imperative language. They use minimal typing for loop-conditionals and expressions which can (possibly) throw exceptions to enforce the termination sensitive form of non-interference, i.e. if the security types of these conditionals and expressions are minimal, in terms of the security level, then the termination behavior will only depend on low variables, thereby ensuring that termination behavior of a program can not leak information. Volpano and Smith prove an additional lemma which states that if all conditionals and expressions that can throw exceptions are typed minimally then the program will be time-sensitive termination-sensitive non-interfering.

The main disadvantage of this approach is that information flow type systems are very restrictive, and reject many secure programs, even for sim-

ple programming languages. In order to overcome this limitation, Barthe, D'Argenio, and Rezk [6] have investigated logical formulations of non-interference that allow a more precise analysis of programs. Such formulations are often sound and complete, and also amenable to interact with automated verification techniques, such as theorem-proving or model-checking.

4 Adding objects, methods and exceptions

4.1 Language

We extend our simple imperative language with objects, methods (without dynamic dispatch) and an exception mechanism as in Java, and we call OO the resulting language. The sets Expr of *expressions* and Comm of *commands* are given by the syntaxes in Figure 5.

An OO program P comes equipped with a set $\mathcal{C}$ of class names, including a class Throwable of exceptions, a set $\mathcal{F}$ of field names, and a set of method declarations of the form $m(\vec{x}) := c$, where m is a method name, $\vec{x}$ is a vector of variables (method formal parameters), and c is a command in Comm. For every program in OO, we distinguish a main method namely main.

$$e ::= x \mid n \mid e_1 \; op \; e_2 \mid e.f \mid \mathsf{new}\ C \mid e.m(\vec{e}) \mid (C)e$$

$$c ::= x := e \mid c_1; c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid$$
$$e.f := e \mid \mathsf{return}\ e \mid \epsilon \mid \mathsf{try}\ c_1\ \mathsf{catch}(\mathsf{Exception\ x})\ \mathsf{c_2} \mid \mathsf{Throw}$$

where *op* is either a primitive operation $+, \times$, a comparison operation $<, \leq, =$, or the (unconditional) boolean connectives | and & and ϵ is the empty program (skip).

Fig. 5. The language OO

The operational semantics of the language is given in Appendix A; due to the presence of exceptions, performing one-step execution of a command may either lead to a normal state, or to an exceptional state. For the sake of simplicity, we assume that the only commands that may raise exceptions in our language are $x.f := e$, $x := e.f$ (where e cannot throw exceptions), and an explicit `Throw` statement.

The operational semantics is used to define an evaluation relation that relates programs, memories, results, and execution times.

4.2 Problem statement

In our extended language, timing leaks may occur due to explicit or implicitly thrown exceptions.

We require that all exceptions that are thrown under a high security level are surrounded by a `try-catch` block. In order to write a correct transformation, we also need to assume that high exceptions (i.e. exceptions thrown by a command containing $e.f$ where $e.f$ is a high expression) are handled in the same method where the exceptions are thrown, i.e. high exceptions cannot be propagated. The same applies to `Throw` commands inside influence of high conditionals.

4.3 The transformation

Figure 6 shows how we extend the translation $\mathcal{T}$ from Figure 4 to allow for objects and exceptions. The transformation $\mathcal{T}$ uses a tail recursive function $\mathcal{T}_1$, showed in Figure 7. Intuitively $\mathcal{T}_1$ transforms every high command that might throw an exception into a set of commands without timing leaks.

In the transformation $\mathcal{T}_1(c_0, c_1, c_2, x)$, argument c_0 is a partial result of sequence of commands that either do not depend of high variables or that depend of high variables but that have been transformed in a sequence of timing-leaks free commands; its second argument c_1 represents commands in the original sequence of commands of a try part of a `try-catch` command that has to be transformed into a sequence of timing-leaks free commands; the third argument c_2 corresponds to the original command in the catch part of a `try-catch` command, and x is its variable. For example, in the case where the first command in c_2 is $x'.f := e$, the 'dummy' object in its transformation, that is created if variable x' holds a null reference, is used to perform the assignment $x'.f := e$, which is then aborted. Symmetrically we also create a dummy object within a transaction-block which is then aborted directly, so that the assignment can be performed on the initial (non-null) object. We use the *static* type of the object to create a corresponding dummy object. In this way we ensure that fields of dummy objects are the same fields of the original one (either directly or via inheritance).

Figure 8 presents an example of how the command $\mathsf{try}\ \{x := 1; y.f := v; z.f := v'\}\ \mathsf{catch}(Exception\ x')\ \{c_3\}$ is tranformed, assuming that x and $z.f$ are low expressions and that $y.f$ is a high expression.

One can extend the results of the previous section and show that the transformation removes timing leaks from non-interfering programs. The proofs are similar to those of the previous section, but they are more involved, because of the increased complexity of the semantics and definitions of non-interference.

4.4 Enforcing termination-sensitive non-interference

In order to yield time-sensitive termination-sensitive non-interfering programs, the transformation must take as inputs non-interfering low-recursive and low-exceptional programs. Establishing termination-*insensitive* non-interference can be done with the type system suggested by Banerjee and Naumann [4,5]. Their language is almost identical to the exception-free fragment of OO, and

$$
\begin{array}{lll}
\mathcal{T}(\mathsf{try}\ c_1\ \mathsf{catch}(\mathsf{Exception}\ x)\ c_2) & = & \mathcal{T}_1(\epsilon, c_1, c_2, x) \\
\mathcal{T}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \\
 & & \texttt{IF}\ \mathsf{sl}(e) = \mathsf{L}\ \texttt{THEN}\ \mathsf{if}\ e\ \mathsf{then}\ \mathcal{T}(c_1)\ \mathsf{else}\ \mathcal{T}(c_2) \\
 & & \texttt{ELSE}\ \ \mathsf{if}\ e\ \mathsf{then}\ \texttt{beginT;}\mathcal{T}(c_2)\texttt{;abortT;} \\
 & & \qquad \texttt{beginT;}\mathcal{T}(c_1)\texttt{;commitT;} \\
 & & \quad \mathsf{else}\ \texttt{beginT;}\mathcal{T}(c_1)\texttt{;abortT;} \\
 & & \qquad \texttt{beginT;}\mathcal{T}(c_2)\texttt{;commitT;} \\
\mathcal{T}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \mathsf{while}\ e\ \mathsf{do}\ \mathcal{T}(c) \\
\mathcal{T}(c_1; c_2) & = & \mathcal{T}(c_1); \mathcal{T}(c_2)
\end{array}
$$

Transformation $\mathcal{T}$ for other commands is defined as the identity.

Fig. 6. Transformation $\mathcal{T}$ for $\mathsf{OO}_{\mathsf{ex}}$

their type system can easily be adopted to such a fragment. In an ongoing collaboration with D. Naumann, we are studying an information flow type system for a fragment of Java with exceptions. Such a type system could be used to check that programs that are submitted to the transformation have the expected properties, so that the transformed programs do not have timing leaks.

5 Observations and Practical concerns

This section describes some general observations about our approach, as well as some problems and possible solutions when it is applied in practice. We describe these practical concerns in the context of the (sequential part of the) programming language Java. However we want to stress that the code translation can be applied to any object oriented language as long as (nested) transaction mechanisms are supported or can be implemented.

5.1 Time-outs

We do not want to use transactions which can time-out, i.e. which have a maximum bound on the time they can take, then time-out and consequently perform an abort. Either one of two things can happen in such a scenario which are both undesirable:

(i) Information can be leaked,

$\mathcal{T}_1\ (c_0, \epsilon, c_3, x) = \mathsf{try}\ c_0\ \mathsf{catch(Exception}\ x)\ c_3$

$\mathcal{T}_1\ (c_0, x' := e.f; c_2, c_3, x) =$

$\mathtt{IF}\ \mathsf{sl}(x') = \mathsf{L}\ \mathtt{THEN}\ \mathcal{T}_1(c_0; x' := e.f, c_2, c_3, x)\ \mathtt{ELSE}\ \mathcal{T}_1(c_0; c_1', c_2, c_3, x)\ \ \textit{where}\ c_1' =$

$\mathsf{if}\ e == \mathsf{null}\ \mathsf{then}$

$\mathsf{beginT}; c_3; \mathsf{commitT}; \mathsf{beginT}; x' := e; x' := (\mathsf{new}\ C).f; \mathsf{abortT};$

$\mathsf{else}\ \mathsf{beginT}; c_3; \mathsf{abortT}; \mathsf{beginT}; x' := \mathsf{new}\ C; x' := e.f; \mathsf{commitT};$

$\mathcal{T}_1\ (c_0, (\mathsf{while}\ e\ \mathsf{do}\ c_1'); c_2, c_3, x) = \mathcal{T}_1(c_0; \mathsf{while}\ e\ \mathsf{do}\ \mathcal{T}_1(\epsilon, c_1', c_3, x), c_2, c_3, x)$

$\mathcal{T}_1\ (c_0, (\mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'); c_2, c_3, x) =$

$\mathtt{IF}\ \mathsf{sl}(e) = \mathsf{L}\ \mathtt{THEN}\ \mathcal{T}_1(c_0; \mathsf{if}\ e\ \mathsf{then}\ \mathcal{T}_1(\epsilon, c_1', c_3, x)\ \mathsf{else}\ \mathcal{T}_1(\epsilon, c_2', c_3, x), c_2, c_3, x)$

$\mathtt{ELSE}\ \mathcal{T}_1(c_0; \mathsf{if}\ e\ \mathsf{then}$

$\mathtt{beginT;}\mathcal{T}_1(\epsilon, c_2', c_3, x)\mathtt{;abortT;beginT;}\mathcal{T}_1(\epsilon, c_1', c_3, x)\mathtt{;commitT;}\ \mathsf{else}$

$\mathtt{beginT;}\mathcal{T}_1(\epsilon, c_1', c_3, x)\mathtt{;abortT;beginT;}\mathcal{T}_1(\epsilon, c_2', c_3, x)\mathtt{;commitT;}, c_2, c_3, x)$

$\mathcal{T}_1\ (c_0, x'.f := e; c_2, , c_3, x) =$

$\mathtt{IF}\ \mathsf{sl}(x'.f) = \mathsf{L}\ \mathtt{THEN}\ \mathcal{T}_1(\epsilon, x'.f := e, c_3, x)\ \mathtt{ELSE}\ \mathcal{T}_1(c_0; c_1', c_2, c_3, x)\ \ \textit{where}\ c_1' =$

$\mathsf{if}\ e == \mathsf{null}\ \mathsf{then}$

$\mathsf{beginT}; c_3; \mathsf{commitT}; \mathsf{beginT}; \mathsf{beginT}; x' := \mathsf{new}\ C; \mathsf{commitT}; x'.f := e; \mathsf{abortT};$

$\mathsf{else}\ \mathsf{beginT}; \mathsf{beginT}; x' := \mathsf{new}\ C; \mathsf{abortT}; c_3; \mathsf{abortT}; \mathsf{beginT}; x'.f := e; \mathsf{commitT}$

$\mathcal{T}_1\ (c_0, e.f := e'; c_2, , c_3, x) = \mathcal{T}_1(c_0; e.f := e', c_2, , c_3, x)\ \textit{where}\ e\ \textit{is not a variable}$

$\mathcal{T}_1\ (c_0, x' := e; c_2, , c_3, x) = \mathcal{T}_1(c_0; x' := e, c_2, , c_3, x)\ \textit{where}\ e\ \textit{is not}\ e'.f$

$\mathcal{T}_1\ (c_0, \mathsf{return}\ e; c_2, c_3, x) = \mathcal{T}_1(c_0; \mathsf{return}\ e, c_2, c_3, x)$

$\mathcal{T}_1\ (c_0, \epsilon; c_2, c_3, x) = \mathcal{T}_1(c_0, c_2, c_3, x)$

$\mathcal{T}_1\ (c_0, \mathsf{throw}; c_2, c_3, x) = \mathcal{T}_1(c_0; \mathsf{throw}, \epsilon, c_3, x)$

$\mathcal{T}_1\ (c_0, \mathsf{try}\ c_1'\ \mathsf{catch(Exception}\ x')\ c_2'; c_2, c_3, x) = \mathcal{T}_1(c_0; \mathcal{T}_1(\epsilon, c_1', c_2', x'), c_2, c_3, x)$

Fig. 7. Transformation $\mathcal{T}_1$ for $\mathsf{OO}_{\mathsf{ex}}$

(ii) The semantics of our language is no longer preserved.

Consider the translation $\mathcal{T}$ from Figure 4 again. Suppose that each transaction-block has a fixed time-out t_{to}. Then if the command `c1` hangs and the command `c2` terminates normally in a time $t_{c2} < t_{to}$ the following situation occurs:

- if `c` evaluates to true both transactions will be aborted, the first (`c2`) explicitly via a call to `abortT` the second (`c1`) implicitly via a time-out (because `c1` does not terminate),
- however if `c` evaluates to false then the first transaction-block will again time-out and thus abort, but the second block will be executed.

So in this scenario information can be leaked (about the value of `c`).

We can prevent this undesirable behavior by putting the complete `if-then-else` in a transaction block, i.e. if `body` is the (translated) if-then-else from Figure 4, we would put this again inside a transaction block, thereby obtaining program fragment `beginT; body; commitT;`. Information is now no longer

$\mathcal{T}(\mathsf{try}\{x := 1; y.f := v; z.f := v'\}\mathsf{catch}(\mathsf{Exception}\ x')\{c_3\}) =$
$\mathcal{T}_1(\epsilon, x := 1; y.f := v; z.f := v', c_3, x') =$
$\mathcal{T}_1(x := 1, y.f := v; z.f := v', c_3, x') =$

```
try
    x := 1;
    if(y == null){
                beginT; T(c3); commitT; }
                beginT;
                   beginT; y := new C; commitT;
                   y.f := v;
                abortT;
    else{
                beginT;
                   beginT; y := new C; abortT;
                   T(c3);
                abortT;
                beginT; y.f := v; commitT; }
    z.f := v';
catch(Exception x')
     c3
```

Fig. 8. Example

leaked because if one of the inner transaction blocks time-outs the outer block will obviously also time-out and thus the whole `if-then-else` will no longer be executed. However this will have the undesirable side-effect that the semantics of our language is no longer preserved.

5.2 Termination

Our transformation may turn a terminating program in a program that hangs, which is clearly undesirable. Non-termination arises in the transformation when we consider program fragments like `if` e `then` $c1$ `else` $c2$ (a similar argument applies to commands that may throw exceptions). If command `c1` terminates normally and `c2` hangs then the translated program will always hang. This behavior results from the fact that the termination mode of a command in a way *overrules* the transaction mechanism. In order to minimize the cases of non-termination and the overhead caused by our transformation, the translation of `if-then-else` blocks with a low conditional is given by the obvious recursive clause. Finer approaches that do not introduce non-termination are left for future work.

5.3 Preventing code explosion

Automatic program transformations may lead to a substantial increase in the size of code, or even to code explosion. Our approach is no exception to this, since the size of the transformed code is exponential in the nested ifs. In order to avoid code explosion, one can implement the transaction mechanism at the *byte-code* level using subroutines. The basic idea there is that conditional branching statements, which are usually compiled as,

1	ifeq i
..	c_2
$i-1$	goto j
i	c_1
	..
j	return

are compiled, if they branch over a high expression, into

1	ifeq 9	13	jsr k_2
2	beginT	14	abortT
3	jsr k_1	15	goto 18
4	abortT	16	return
5	beginT	k_1	store x
6	jsr k_2		c_1'
7	commitT		...
8	goto 18		ret x
9	beginT	k_2	store x
10	jsr k_1		c_2'
11	commitT		...
12	beginT		ret x

where the instruction jsr k calls the subroutine starting at k.

Another method that can be used to minimize code explosion is the use of *assertions*. The method is complementary, in that it aims at reducing the branching at instructions that may raise an exception, but in fact do not. Consider the following code fragment:

```
// assert o1!=null
   o1.f = c;
```

The assertion states that the variable `o1` is not a non-null reference. So we do not have to encapsulate this assignment in a try-catch block and proceed via

the translation as shown in Figure 6. That is, provided the assertion is true, which has to be checked with a separate tool such as ESC/Java2 [8]. Note that assertions can also be used to improve the precision of information flow type systems.

5.4 Timing model

Our operational semantics adopts a very simple model of time. In particular, the execution time of arithmetic expressions is constant and independent of the values assigned to variables. Furthermore, our model of execution time does not reflect the fact that the execution time for a given state may vary depending on the execution history of the program. Thus mechanisms such as caching interfere with our approach and could allow timing leak.

One drastic solution is to turn off all caching features, such as data-cache, instruction-cache and virtual memory. A better solution would be to impose appropriate interactions between caching and transaction mechanisms. This is left for future work.

5.5 Optimizing and JIT-compiling

Although we prove that the transformation at the source code level removes all timing leaks we need to be careful when compiling and running the transformed program. In the case of Java we have to ensure that no optimizations are performed when compiling to byte-code. A 'smart' compiler probably removes code fragments like `beginT; ... ; abortT` since semantically these are equivalent to a `skip` statement. In our approach it is crucial that these code-blocks are also present in the compiled byte-code program. Other forms of optimization can have similar undesirable results.

Java byte-code is interpreted by a virtual machine. All modern Java virtual machines use, so called, just-in-time (JIT) compilation of byte-code. This means that code is compiled 'on the fly' and that calls to the same method with the same parameters can take different execution times[4] (if the code needs to be compiled on the fly or not). If no timing leaks are allowed then this JIT-compilation has to be disabled and the byte code needs to be interpreted without optimization.

5.6 Nested Transactions in Java

As far as we know there is no API that implements nested transactions for the Java Standard Edition. Both the Java Enterprise Edition and the Java tailored for Smart Cards, Java Card [7], have transaction mechanisms. The former in the form of the the Java Transaction API[5] and the latter has transactions

[4] Assuming the method is evaluated in the same state.
[5] http://java.sun.com/products/jta/

as a core language feature. However both transaction mechanisms only allow non-nested transactions.

The one actual implementation of a nested transaction mechanism (for a language of the Java family) we are aware of is the implementation by Lecomte, Grimaud and Donsez [14] for Java Card. In order to empirically establish how well our proposed approach works we need to implement a nested transaction mechanism [16]. We leave this as future work.

6 Conclusions and Future Work

We show that, under certain assumptions, using nested transactions to remove timing leaks from sequential object oriented programs is feasible. In future work we want to implement our approach and provide some empirical support for our work as well as give a better estimate of overhead. Another future direction of research is to see if we can use transaction mechanisms to enforce non-interference in a multi-threaded environment.

References

[1] J. Agat. Transforming out Timing Leaks. In *27th ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, January 2000.

[2] J. Agat. *Transforming out Timing Leaks in Practice*, chapter II from [3]. Department of Computing Science Chalmers University of Technology and Göteborg University, September 2000.

[3] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Department of Computing Science Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, December 2000.

[4] A. Banerjee and D. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proc. of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267. IEEE Computer Society Press, June 2002.

[5] A. Banerjee and D. Naumann. Stack-Based Access Control for Secure Information Flow. *Journal of Functional Programming*, 200x. Special Issue on Language-Based Security, To appear.

[6] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.

[7] Z. Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley, June 2000.

[8] D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M .Huisman, J-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2004.

[9] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Comp. Soc. Press, 1982.

[10] D. Hedin and D. Sands. Timing Aware Information Flow Security for a JavaCard-like Bytecode. In F. Spoto, editor, *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE) proceedings*, ENTCS, pages 149–166. ELsevier, 2005.

[11] P.C. Kocher. Timing attacks on implementations of diffie-helman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[12] B. Köpf and Heiko Mantel. Eliminating timing leaks by unification (extended abstract), 2004.

[13] B.W. Lampson. A note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[14] S. Lecomte, G. Grimaud, and D. Donsez. Implementation of Transactional Mechanisms for Open SmartCard. In *GEMPLUS Developer Conference*, 1999.

[15] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[16] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, 1981.

[17] A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on selected areas in communications*, 21(1), 2003.

[18] D. Volpano and G. Smith. Eliminating Covert Flows with Minimum Typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168. IEEE Computer Society Press, June 1997.

A Operational Semantics

In this section we give the timed operational semantics of OO. The set of values of OO is defined as $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations, $x \in \mathcal{X}$, where $\mathcal{X}$ is a set of local variables, $n \in \mathbb{Z}$ and o is an object from a set, namely $\mathcal{O}$.

The set State of OO states is defined as the set of pairs $\langle sf, h \rangle$ where sf is a stack of frames, and h is a heap.

A frame is of the form $\langle c, \rho \rangle$ where c is in Comm, ρ is a mapping from local variables from a set of variables $\mathcal{X}$ to values. We distinguish a special variable res to store results of execution of programs.

Heaps are modeled as a partial function $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{F} \rightharpoonup \mathcal{V}$, i.e. as the set of finite functions from $\mathcal{F}$ to $\mathcal{V}$. We let Heap be the set of heaps and R be the set of all variable mappings. We further use $\sqsubseteq : \mathcal{C} \to \mathcal{C} \to \mathbb{B}$ to denote subclass relation and the functions $\mathsf{static} : \mathcal{O} \to \mathcal{C}$ and $\mathsf{dynamic} : \mathcal{O} \to \mathcal{C}$ give the static and dynamic type of an object respectively, the function $\mathsf{cdynamic} : \mathcal{C} \times \mathcal{O} \to \mathcal{O}$ assigns a (new) dynamic type to an object. Furthermore we have an allocator function $\mathsf{fresh} : \mathsf{Heap} \times \mathcal{C} \to \mathcal{L}$ and a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$ which puts a new object on the heap. The operational semantics of OO can be found in Figure A.1. Note that we assume that the object `this` remains unmodified during the execution of a program.

The relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq T \times \mathsf{State} \times ((\epsilon \cup \mathcal{V}) \times \mathsf{R} \times \mathsf{Heap})$ formalizes –besides the operational semantics–the *execution time* of OO. Figure A.1 shows both *quantified* variables, such as h, or ρ, and *constants*, such as t. The set of constants includes constant executions times, namely t_{R_1}, t_{R_2}, t_{OP}, $t_{:=}$, t_W, t_N, t_C , for different commands and operations.

We extend the semantics of OO with exceptions in Figure A.2. The exceptional states of the form $< exc > s$ where $s \in \mathsf{State}$ are used for propagation of exceptions, that is every time that a subexpression or subcommand is evaluated to a state of the form $< exc > s$, this state is propagated to the containing expression or command, or in the case of method calls, if a method evaluates to an exceptional states, the exception is propagated to the caller method.

$$\frac{}{\langle x,\rho\rangle :: fs, h \rightsquigarrow_{t_{R_1}} \langle \rho(x),\rho\rangle :: fs, h} \qquad \frac{h(o).f = v}{\langle o.f,\rho\rangle :: fs, h \rightsquigarrow_{t_{R_2}} \langle v,\rho\rangle :: fs, h}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle e',\rho\rangle :: fs, h'}{\langle e.f,\rho\rangle :: fs, h \rightsquigarrow_t \langle e'.f,\rho\rangle :: fs, h'} \qquad \frac{v \ op \ v' = v''}{\langle v \ op \ v',\rho\rangle :: fs, h \rightsquigarrow_{t_{OP}} \langle v'',\rho\rangle :: fs, h}$$

$$\frac{\langle e_1,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_1',\rho\rangle :: fs, h'}{\langle e_1 \ op \ e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_1' \ op \ e_2,\rho\rangle :: fs, h'} \qquad \frac{\langle e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_2',\rho\rangle :: fs, h'}{\langle v \ op \ e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle v \ op \ e_2',\rho\rangle :: fs, h'}$$

$$\frac{}{\langle x := v,\rho\rangle :: fs, h \rightsquigarrow_{t_:=} \langle \epsilon, \rho \oplus \{x \mapsto v\}\rangle :: fs, h} \qquad \frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle e',\rho\rangle :: fs, h'}{\langle x := e,\rho\rangle :: fs, h \rightsquigarrow_t \langle x := e',\rho, h'\rangle}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle e',\rho\rangle :: fs, h'}{\langle x := e.m(\vec{e}),\rho\rangle :: fs, h \rightsquigarrow_t \langle x := e'.m(\vec{e}),\rho\rangle :: fs, h'}$$

$$\frac{\langle \vec{e},\rho\rangle :: fs, h \rightsquigarrow_t \langle \vec{e'},\rho\rangle :: fs, h'}{\langle x := o.m(\vec{e}),\rho\rangle :: fs, h \rightsquigarrow_t \langle x := o.m(\vec{e'}),\rho\rangle :: fs, h'}$$

$$\frac{m(\vec{x}) := c}{\langle x := o.m(\vec{v}),\rho\rangle :: fs, h \rightsquigarrow_t \langle c, \vec{x} \mapsto \vec{v}\rangle :: \langle x := res,\rho\rangle :: fs, h}$$

$$\frac{\langle c_1,\rho\rangle :: fs, h \rightsquigarrow_t \langle c_1',\rho'\rangle :: fs, h'}{\langle c_1; c_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle c_1'; c_2,\rho'\rangle :: fs', h'}$$

$$\frac{\langle x := o.m(\vec{v}),\rho\rangle :: fs, h \rightsquigarrow_t \langle c,\rho'\rangle :: \langle x := res,\rho\rangle :: fs, h'}{\langle x := o.m(\vec{v}); c_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle c,\rho'\rangle :: \langle x := res; c_2,\rho'\rangle :: fs', h'}$$

$$\frac{}{\langle \epsilon; c_2,\rho\rangle :: fs, h \rightsquigarrow_0 \langle c_2,\rho'\rangle :: fs, h'}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle true,\rho\rangle :: fs, h'}{\langle \textsf{while } e \textsf{ do } c,\rho\rangle :: fs, h \rightsquigarrow_t \langle c; \textsf{while } e \textsf{ do } c,\rho\rangle :: fs, h'} \qquad \frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle false,\rho\rangle :: fs, h'}{\langle \textsf{while } e \textsf{ do } c,\rho\rangle :: fs, h \rightsquigarrow_t \langle \epsilon,\rho\rangle :: fs, h'}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle true,\rho\rangle :: fs, h'}{\langle \textsf{if } e \textsf{ then } c_1 \textsf{ else } c_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle c_1,\rho\rangle :: fs, h'}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle false,\rho\rangle :: fs, h'}{\langle \textsf{if } e \textsf{ then } c_1 \textsf{ else } c_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle c_2,\rho\rangle :: fs, h'}$$

$$\frac{o \in dom(h) \qquad f \in dom(h(o))}{\langle o.f := v,\rho\rangle :: fs, h \rightsquigarrow_{t_W} \langle \epsilon, \rho, h \oplus \{o \mapsto h(o) \oplus \{f \mapsto v\}\}\rangle} \qquad \frac{\langle e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_2',\rho, h'\rangle}{\langle o.f := e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle o.f := e_2',\rho, h'\rangle}$$

$$\frac{\langle e_1,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_1',\rho, h'\rangle}{\langle e_1.f := e_2,\rho\rangle :: fs, h \rightsquigarrow_t \langle e_1'.f := e_2,\rho\rangle :: fs, h'} \qquad \frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle e',\rho\rangle :: fs, h'}{\langle \textsf{return } e,\rho\rangle :: fs, h \rightsquigarrow_t \langle \textsf{return } e',\rho, h'\rangle}$$

$$\frac{}{\langle \textsf{return } v,\rho\rangle :: \langle c,\rho\rangle :: fs, h \rightsquigarrow_{t_R + t_:=} \langle c, \rho \oplus \{res \mapsto v\}\rangle :: fs, h} \qquad \frac{}{\langle \textsf{return } v,\rho\rangle :: \epsilon, h \rightsquigarrow_{t_R} \langle v,\rho\rangle :: fs, h}$$

$$\frac{o = \textsf{fresh}(h, C)}{\langle \textsf{new } C,\rho\rangle :: fs, h \rightsquigarrow_{t_N} \langle o,\rho\rangle :: fs, h \oplus \{o \mapsto \textsf{default}_C\}}$$

$$\frac{\textsf{dynamic}(o) \sqsubseteq C}{\langle (C)o,\rho\rangle :: fs, h \rightsquigarrow_{t_C + t_W} \langle o,\rho\rangle :: fs, h' \oplus \{o \mapsto \textsf{cdynamic}(C)\}}$$

$$\frac{\langle e,\rho\rangle :: fs, h \rightsquigarrow_t \langle e',\rho\rangle :: fs, h'}{\langle (C)e,\rho\rangle :: fs, h \rightsquigarrow_t \langle (C)e',\rho\rangle, h'}$$

Fig. A.1. Small Step Operational Semantics for the core language OO

$$\frac{\langle e_1, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle e_1 \ op \ e_2, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s, h'}$$

$$\frac{\langle e_2, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle v \ op \ e_2, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

$$\frac{\langle e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle (C)e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

$$\frac{\langle e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle e.f, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

$$\frac{\langle e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs', h'}{\langle x := e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

$$\frac{\langle e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle \mathsf{return}\ e, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

$$\frac{o = \mathsf{fresh}(h', \mathsf{Throwable})}{\langle \mathsf{null}.f := e_2, \rho\rangle :: fs, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{o = \mathsf{fresh}(h', \mathsf{Throwable})}{\langle \mathsf{null}.m(\vec{e}), \rho\rangle :: fs, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h' \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{o = \mathsf{fresh}(h, \mathsf{Throwable})}{\langle \mathsf{Throw}, \rho\rangle :: fs, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{\langle c_1, \rho\rangle :: fs, h \rightsquigarrow_t < exc > \langle o, \rho'\rangle :: fs, h'}{\langle \mathsf{try}\{c_1\}\ \mathsf{catch}(\mathsf{Exception\ x})\{c_2\}, \rho\rangle :: fs, h \rightsquigarrow_{t+t_{:=}} \langle c_2, \rho' \oplus \{x \mapsto o\}\rangle :: fs, h'}$$

$$\frac{\langle c_1, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}{\langle c_1; c_2, \rho\rangle :: fs, h \rightsquigarrow_t < exc > s :: fs, h'}$$

Fig. A.2. Operational Semantics for OO