In 19th IEEE Computer Security Foundations Workshop, CSFW'06, Venice 2006.

Noninterference in the presence of non-opaque pointers

Daniel Hedin David Sands
Chalmers University of Technology, Sweden

Abstract

A common theoretical assumption in the study of information flow security in Java-like languages is that pointers are opaque – i.e., that the only properties that can be observed of pointers are the objects to which they point, and (at most) their equality. These assumptions often fail in practice. For example, various important operations in Java's standard API, such as hashcodes or serialization, might break pointer opacity. As a result, information-flow static analyses which assume pointer opacity risk being unsound in practice, since the pointer representation provides an unchecked implicit leak. We investigate information flow in the presence of non-opaque pointers for an imperative language with records, pointer instructions and exceptions, and develop an information flow aware type system which guarantees noninterference.

1 Introduction

Background The area of *information flow security* deals with different means to define and enforce information flow policies. Such policies describe the secrecy levels of data, and how data may flow through the system. Semantically, it is usual to define the meaning of "flow" as an end-to-end condition usually referred to as *noninterference*. Given a policy which partitions inputs and outputs into secret (not observable by the attacker) and public (observable by the attacker), noninterference demands simply that the public *outputs* are independent of the secret *inputs*.

A popular way of enforcing noninterference in programs is to equip the language with a *security type system*, i.e., a type system which tracks information flow through a program by security annotations in the types. Typically, a security type system has the property that well typed programs are noninterfering. For further information on the topic we refer the reader to Sabelfeld and Myers' comprehensive overview of the area [10].

To date there are many formulations of noninterference, depending, for instance, on the properties of the underlying language, whether the attacker can observe timing properties, (the absence of) termination, and what parts are considered to be publicly observable.

In particular, a fair amount of work regarding information flow security has been conducted for Java and Java-like languages, ranging from more theoretical work such as Banerjee and Naumann's study [2], to the more practical work on complete systems such as JIF [8], which is a full-scale implementation of an information flow type system for a security-typed extension of Java. For a mainstream language Java is arguably a reasonable choice for such a study since the Java core language is, in principle, fairly small and clean.

The Problem Targeting real languages is beneficial: your work may have practical impact and existing implementations (often on many platforms) relieve you from substantial implementation work. While enjoying those benefits, analysing an existing language also means that one has to be faithful to the implementations. In the case of Java, this extends beyond the core language to the API, some of which is native and cannot be implemented in Java. However clean and concise the core language and its theoretical abstractions may be, the runtime environment and native methods can break many abstractions that are typically assumed in both theoretical and practical studies of secure information flow. When abstractions are broken, attacks are possible. By way of illustration we present such an attack on JIF showing a well-typed declassification-free program that leaks its secret.

As an example of a problematic native method, consider the following program, which is using the default implementation of the method **toString** found in the **Object** class.

```
public class B {
    public static void main(String[] ss) {
        boolean secret = Boolean.getBoolean(ss[0]);
        if (secret) new A();
        System.out.println(new Object());
    }
}
class A { }
```

Apart from printing the name of the class of the object,

the **toString** method prepends an integer to the name. The value of that integer, and thus the result of executing this method can be affected by the initial allocation of an object of a previously unused class. When run, this program will deterministically[1] give different outputs depending on the value of secret. In this case (since the secret is a boolean) the value of the secret can be completely determined from the output. This is referred to as an *implicit flow*; implicit flows arise when changes in the environment are indirectly controlled by a secret value, thus encoding information about the secret.

This program can be translated into JIF[2] in a straightforward way such that the variable **secret** is secret and the output is performed on a public channel. Even though the JIF version does not use any of JIF's declassification mechanisms, the program is accepted by the JIF compiler while retaining semantic behaviour, which means that the noninterference is broken. (The JIF program is not presented here for space reasons).

A more important example, from the point of view of Java, is the use of built-in hashcodes. These can be used in a manner similar to the above:

```
public class A {
    public static void main(String[] ss) {
        HashMap m = new HashMap();
        boolean secret = Boolean.getBoolean(ss[0]);

        Object o = new Object();
        if (secret) { m.put(o,o); }
        System.out.println((new Object()).hashCode());
    }
}
```

The hashcode of the newly allocated object will be deterministically different depending on whether the body of the secret conditional is run or not. The reason for the difference is that the hashcodes are drawn from one global sequence, and that the hashcode needed to put the object into the hashmap is allocated when needed, i.e., in the body of the secret conditional in this example.

The difference will typically be deterministic, even though the sequence is normally generated by a pseudo random generator, since for hashing, it is only the distribution that is important, not the randomisation, which means that the same sequence is used in every run.

Non-Opaque Pointers The above examples illustrate that there is a risk that the runtime environment *extends the semantics* of the core language, in such a way that assumptions on the core language are broken or new covert flows are introduced.

In this paper we focus on one common theoretical assumption that may fail in practice: that pointers are *opaque* – i.e., that the only properties of pointers that can be observed are the objects to which they point, and (at most) the equality of the pointers.

Java is typically assumed to have opaque pointers; although it has pointer equality, the only pointer constant available is the nil pointer and operations like coercing a pointer value to an integer are not present in the core language. However, we have no guarantee that the API does not contain methods that reveal pointer values and breaks the opaqueness. To connect to the above examples, a cheap way of implementing hashcodes is to simply use the pointer value — a plausible implementation for a JRE running on a platform where resources are scarce[3].

If a language does not have opaque pointers and is equipped with deterministic allocation, the side effects of allocation must be modeled to prevent leaks of this kind. Even though non-deterministic allocation would prevent implicit leaks via allocation this is *not an option* if we want to make use of existing implementations that are deterministic.

Our Contribution We present a security type system for a small imperative language with exceptions chosen to capture the essence of the problem in languages similar to Java.

The type system is based on the realisation that operations on pointers are *opaque with respect to some of the properties of the pointer* and works by adding additional security and domain annotations to the pointer type; in particular we differentiate the security type of pointer values from the security type associated with the record pointed to.

Other novel features of the type system include the combination of *flow sensitivity*, meaning that variables are not required to have a fixed security level, and *exceptions*. Pointers pose problems in that most operations on pointers are partial — they fail if applied to the nil pointer. To be able to use secret pointers (pointers whose representation is influenced by a secret) in a system where exceptions are considered observable, the system includes the ability to rule out information flows through nil-pointer exceptions by tracking the domain of a pointer location in the types.

Related Work Parts of the present paper build on some of the technical development from Banerjee and Naumann's study of noninterference for a Java-like language [2]. The language studied there contains the assumption that pointers are opaque, and introduces a formulation of noninterference for heap structures via a bijection on pointers for the parts

[1] Using the specific implementations that we tested: JRE 1.5.0_01b08, SunOS 5.9 and JRE 1.4.2_09232, Mac OS X

[2] jifc version 1.1.1,jifc version 2.0.0

[3] In fact, net rumor has it that in the early JREs – before the generational garbage collector – this was in fact the implementation

of the heap reachable from low security data. This assumption and the use of a bijection has been adopted in recent work on noninterference for Java or Java bytecode [4, 6, 7]. Despite the fact that in the semantics for a language with non-opaque pointers we cannot identify heaps up to bijective renaming, we show that it is still possible to adapt the bijection-style formulation of noninterference.

Outline Section 2 provides an overview of key ideas in the type system. Section 3 defines the syntax and semantics of the language. Section 4 defines the types for values, expressions and statements ending with some highlighting examples. Section 5 introduces the semantic security condition and discusses the correctness proof. Section 6 discusses further work; in particular we consider a type directed transformation for safe object identifiers. Finally, Section 7 concludes.

2 Types for Non-Opaque Pointers

Our goal is an information flow aware type system that correctly handles the problems of non-opaque pointers. In this section we introduce the key ideas which enable us to achieve this, before presenting the actual formal details.

For simplicity we consider a standard two-point lattice of security levels, representing public and secret information. Each environment location, i.e., variables and records, is assigned a security level, indicating whether the location contains secret or public information. The type system then tracks information flows and prevents *dangerous flows*, i.e., flows of secrets into public locations.

There are two different types of information flows: explicit flows — directly copying from a secret to a public location — and implicit flow — differences in public side effects depending on secret values. Implicit flows may arise when the control flow is controlled by secrets. As is standard, following Denning's original approach to analysing programs for secure information flow [5], in order to prevent implicit flows the notion of *security context* is defined. The security context of a program point is the least upper bound of the security levels of the conditional expressions of the enclosing conditionals and while loops. The security context is sometimes referred to as the security level of the program counter, since branching is essentially a conditional update of the program counter. In this work we adopt the standard approach to preventing implicit flows by banning side effects on public data in secret contexts.

Values of Pointers On the language side we will assume a simple means of converting a pointer e to an integer using the coercion expression $(e : \mathsf{int})$. It is this representative non-opaque operation that models that there is some hidden part of the environment, namely *the state of the memory allocator*. This provides an additional source of information flow, and thus we record the security level of the *pointer context*. This tracks the implicit information flow to the memory allocator: when a pointer is allocated in a high context, all subsequent allocated pointers can potentially, via the non-opaque coercion to integers, leak secrets. The pointer context thus affects the security levels of the future pointers.

That the values of newly allocated pointers become secret after the first allocation in a secret context is not as problematic as it may sound, since *most* operations on pointers are opaque to the value of the pointer. For instance, consider the following program assuming that A contains a public integer field f, the variables $h1$, $h2$ and $h3$ are secret and that b is a public pointer to a record of type A,

if (h1) {h3 := **new** A{};} h2 := **new** A{}; b.f := h2.f

which is to be considered secure, even though the value of the pointer stored in $h2$ encodes information about $h1$.

To make use of this we differentiate between the security level of the *value* of the pointer and the security level of the *record to which it points*.

However, as we will see, the type system is *exception sensitive*: abnormal termination and the cause of the termination is considered observable. Most operations on pointers are partial in the sense that they cause a nil-pointer exception if applied to the nil pointer. This means that we cannot freely use partial instructions in secret contexts or with secret data. Consider the following program under the same assumptions as above,

h2 := nil ; **if** (h1) {h2 := **new** A{};} h2.f = 1

which is clearly insecure in an exception sensitive setting; the last instruction fails only if $h1 = false$. Even though this program is to be considered wrong, crafting the type system to rule out programs of this kind faces the risk of ruling out the above secure example as well. We handle this by keeping track of the domain of the pointers in the types by having a type for definitely non-nil pointers. Thus, the type system will allow the topmost example since it will be able to see that $h2$ cannot contain a nil pointer, and thus will never fail.

Highlighting the Pointer Type Annotations The following examples are aimed at highlighting some key properties of our type system by contrasting it with two hypothetical "standard" systems: systems that do not separate the security level of the pointer from the security level of the record pointed to. The first system, which we will refer to as the *unsafe standard system*, does *not* model the implicit leak through secret allocation – i.e., it assumes (incorrectly) that pointers are opaque. The second system, referred to as the

expressions	e	::=	$n \mid \mathsf{nil} \mid e_1 \bullet e_2 \mid e_1 = e_2 \mid (e : \mathsf{int}) \mid (A)\ e \mid e.f \mid x$
left values	lv	::=	$x \mid x.f$
declarations	D	::=	$\mathsf{rec}\ A\ =\ \pi \mid \tau\ x$
handler	h	::=	$\mathsf{catch}(err)\ S$
statements	S	::=	$\mathsf{while}\ e\ S \mid \mathsf{if}\ e\ S_1\ S_2 \mid lv := e \mid S_1; S_2 \mid \mathsf{skip} \mid \mathsf{try}\ S\ h_1 \ldots h_n$
		|	$x := \mathsf{new}\ A\{f_1 = x_1, \ldots, f_n = x_n\}$
program	P	::=	$D_1, \ldots, D_n, S$

Table 1. Syntax

safe standard system, is a similar system that *does* model the implicit leak, but still only has a single security level for pointers. For obvious reasons, the safe standard system is much more restrictive than the unsafe standard system. Our system is less restrictive that the safe standard system, frequently achieving the freedom of the unsafe system. Consider the following program, where $h1$ and $h2$ are secret:

if (h1) {h2 := **new** A;} a := **new** A; a.f = 1

An unsafe standard system would allow a to be public and consider the above program secure. A safe standard system would demand that a is secret and would reject the above program, since the field f is public. Our system would see that it is only the pointer value stored in a that is secret and that a cannot contain the nil pointer. Thus, our system would consider the above example as secure.

The difference between an unsafe standard system and our system is highlighted by the following program, where $h1$ and $h2$ are secret and b is a public pointer to a record of type A.

if (h1) {h2:=**new** A;} a = **new** A; b.f = (a : **int**)

This program would be considered secure by the unsafe standard system even though the pointer value of a, which is written to the public field $b.f$, reflects the secret h. Our system, however, would reject this program.

Before we can describe the type system we need to introduce the specifics of the language and its memory model.

3 The Language

We define a small imperative language with iteration, choice, assignment and sequencing, designed to capture the problems of non-opaque pointers in a language like Java. For simplicity we refrain from modeling features specific to the object orientation and use records and record subtyping to model objects. Furthermore, we add a special coercion expression that models the possibility for the API to break the opaqueness assumption. In this section we introduce the syntax and semantics of the language.

Syntax The syntax of the language is defined in Table 1. Since the syntax is depending on the type language for the declaration of records and variables we begin by introducing the required parts of the type language. A more thorough explanation of the types is found in Section 4 below.

sec. levels	σ	::=	$\mathsf{L} \mid \mathsf{H}$
pointer value	$\mathbf{v}$	::=	$\top \mid \bot$
prim types	τ	::=	$\mathsf{int}\ \sigma \mid A^{\mathbf{v}}\ \sigma_1\ \sigma_2$
record types	π	::=	$\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$

There are two security levels representing secret (H) and public (L) respectively. The primitive types are integers and pointers. Integers are simply annotated with a security level. Let A range over the set of record identifiers $RecID$. Pointer types ($A^{\mathbf{v}}\ \sigma_1\ \sigma_2$) carry a security level for the pointer itself (σ_1), a record security level annotation (σ_2), the record identifier of the record pointed to (A), and a value annotation $\mathbf{v}$ expressing if the pointer is the nil pointer ($\bot$) or not ($\top$). A record type is a collection of uniquely named primitive types, with the names drawn from the set of record field identifiers. Let f range over the record field identifiers.

The expressions are built up by the standard constructs: integer constants, ranged over by n, and the distinguished constant nil, representing the nil pointer, variable names, $x \in Var$, binary operators, $e_1 \bullet e_2$ and field projections, $e.f$. The nil pointer is the only pointer that can be introduced as a constant. Furthermore, the expressions are extended with a non-standard coercion expression, $(e : \mathsf{int})$, which converts a pointer to an integer.

The syntax of the statement language is entirely standard apart from the record allocation, $x := \mathsf{new}\ A\{f_1 = x_1, \ldots, f_n = x_n\}$ which is the only source of pointers beside the nil constant above. Record allocation will never return the nil pointer. The variable x, which is assigned the newly allocated pointer, is available locally in the field assignments in the body of the *new* to allow for the construction of cyclic records,which — more importantly — provides the possibility of creating recursive records without ever introducing nil fields at an intermediate step.

Record identifiers and variables must be declared at the beginning of the program, and their types may be mutually

$$\text{(S-VAR)}\ \frac{E(x) = v}{\langle E, x\rangle \Downarrow v} \qquad \text{(S-INT)}\ \frac{}{\langle E, n\rangle \Downarrow n} \qquad \text{(S-NIL)}\ \frac{}{\langle E, \mathsf{nil}\rangle \Downarrow \mathsf{nil}} \qquad \text{(S-BINOP)}\ \frac{\langle E, e_1\rangle \Downarrow n_1 \quad \langle E, e_2\rangle \Downarrow n_2}{\langle E, e_1 \bullet e_2\rangle \Downarrow n_1[\bullet]n_2}$$

$$\text{(S-EQP)}\ \frac{\langle E, e_1\rangle \Downarrow p_1 \quad \langle E, e_2\rangle \Downarrow p_2}{\langle E, e_1 = e_2\rangle \Downarrow p_1[=]p_2} \qquad \text{(S-PRJ-1)}\ \frac{\langle E, e\rangle \Downarrow p \quad (f = v) \in E(p)}{\langle E, e.f\rangle \Downarrow v} \qquad \text{(S-PRJ-2)}\ \frac{\langle E, e\rangle \Downarrow \mathsf{nil}}{\langle E, e.f\rangle \Downarrow \mathsf{np}}$$

$$\text{(S-CAST-1)}\ \frac{\langle E, e\rangle \Downarrow p \quad T(p, E) <: A}{\langle E, (A)\ e\rangle \Downarrow p} \qquad \text{(S-CAST-2)}\ \frac{\langle E, e\rangle \Downarrow \mathsf{nil}}{\langle E, (A)\ e\rangle \Downarrow \mathsf{nil}} \qquad \text{(S-EERR)}\ \frac{\langle E, e\rangle \Downarrow err}{\langle E, Q[e]\rangle \Downarrow err}$$

$$\text{(S-CAST-3)}\ \frac{\langle E, e\rangle \Downarrow p \quad not(T(p, E) <: A)}{\langle E, (A)\ e\rangle \Downarrow \mathsf{cc}} \qquad \text{(S-COERCE)}\ \frac{\langle E, e\rangle \Downarrow p}{\langle E, (e : \mathsf{int})\rangle \Downarrow coerce(p)}$$

Table 2. Semantics of Expressions

recursive. The declaration assigns a type to the declared entity; record identifiers are assigned record types and variables are assigned primitive types. Finally, a program is a sequence of declarations followed by a statement, which constitutes the body of the program.

Semantics The semantics is a big-step operational semantics with evaluation contexts[11] used to propagate exceptions. Let p range over the set of pointers Ptr.

values	v	::=	$n \mid p \mid \mathsf{nil}$
records	r	::=	$\{A, f_1 = v_1, \ldots, f_n = v_n\}$
pointer contexts	η	::=	(p, n)
environments	E	::=	$\gamma; \rho; \eta$

The primitive values, ranged over by $v \in Val$, are the pointers and the integers. Similar to the record types, the records, ranged over by $r \in Rec$, are collections of named values together with a record identifier. We write $(f = v) \in r$ when $f = v$ is defined in r. The pointer contexts, ranged over by η, are used by the memory model (defined below), and represent the pointer and the size of the most recently allocated record. The environments, ranged over by $E \in Env$, are triples consisting of a variable environment, a heap, and a pointer context. The variable environments, ranged over by γ, and heaps, ranged over by ρ, are partial functions from variables to values and pointers to records respectively. For an environment E we write $E(p)$ to mean $\rho(p)$ and $E(x)$ to mean $\gamma(x)$.

For the treatment of exceptions we extend the values and the environments to include distinguished elements that represent erroneous computation: cc representing a class cast exception, and np representing a nil-pointer exception. Let err range over the set $Err = \{\mathsf{cc}, \mathsf{np}\}$. Let $\hat{v}$ range over the extended values $\widehat{Val} = Val \cup Err$ and $\hat{E}$ range over the extended environments $\widehat{Env} = Env \cup (Err \times Env)$.

The Allocation Model To model the effect of non-opaque pointers we need to model pointer values, and the way in which they are chosen when a record is allocated. Suppose we were to fix on the simplest possible model, e.g. that pointers are natural numbers and that allocation simply picks the next number in sequence. For such an implementation an attacker could potentially learn something only from the order in which records are allocated. But if the actual implementation chose the next pointer according to the *size* of the previously allocated record then the attacker could potentially learn more. Rather than attempting to find a "worst case" model, the approach we take here is not to fix a particular allocator but to work with an abstract model and show that the approach is sound for *any* instantiation.

Definition 3.1 (Allocation Model). *The abstract pointer model is a quadruple $\langle Ptr, live, next, coerce\rangle$, where*

- *Ptr is the set of pointer values,*
- *$live$ is a function which given a heap and a record environment computes a set no smaller than the syntactically live pointers, i.e., those reachable from the environment, and*
- *$next$ is a function which when given the current pointer context η (i.e., the most recently allocated pointer and the size $|r|$ of the corresponding record r) and the current set of live pointers (L), returns the next pointer to be allocated. The function satisfies:*

$$next(\eta, L) \notin L.$$

- *a function $coerce \in Ptr \to \mathbb{Z}$ which models the action of the non-opaque operation. Note that $coerce(\cdot)$ need not be injective (useful, for example, to model hashcodes).*

The model is sufficiently general to cover deterministic garbage collection, since it can allocate a previously allocated but currently dead pointer.

$$\text{(S-WHILE-1)}\ \frac{\langle E, e\rangle \Downarrow 0}{\langle E, \mathsf{while}\ e\ S\rangle \Downarrow E} \qquad \text{(S-WHILE-2)}\ \frac{\langle E, e\rangle \Downarrow n \quad n \neq 0 \quad \langle E, S; \mathsf{while}\ e\ S\rangle \Downarrow \hat{E}_1}{\langle E, \mathsf{while}\ e\ S\rangle \Downarrow \hat{E}_1}$$

$$\text{(S-IF-1)}\ \frac{\langle E, e\rangle \Downarrow 0 \quad \langle E, S_2\rangle \Downarrow \hat{E}'}{\langle E, \mathsf{if}\ e\ S_1\ S_2\rangle \Downarrow \hat{E}'} \qquad \text{(S-IF-2)}\ \frac{\langle E, e\rangle \Downarrow n \quad n \neq 0 \quad \langle E, S_1\rangle \Downarrow \hat{E}'}{\langle E, \mathsf{if}\ e\ S_1\ S_2\rangle \Downarrow \hat{E}'}$$

$$\text{(S-ASSV)}\ \frac{\langle E, e\rangle \Downarrow \langle E', v\rangle}{\langle E, x := e\rangle \Downarrow E'[x \mapsto v]} \qquad \text{(S-ASSF-1)}\ \frac{\langle E, e\rangle \Downarrow \langle E', v\rangle \quad E'(x) = p \quad r = E'(p)}{\langle E, x.f := e\rangle \Downarrow E'[p \mapsto r[f \mapsto v]]}$$

$$\text{(S-ASSF-2)}\ \frac{E(x) = \mathsf{nil}}{\langle E, x.f := e\rangle \Downarrow \mathsf{np}, E} \qquad \text{(S-SEQ)}\ \frac{\langle E, S_1\rangle \Downarrow E_1 \quad \langle E_1, S_2\rangle \Downarrow \hat{E}_2}{\langle E, S_1; S_2\rangle \Downarrow \hat{E}_2} \qquad \text{(S-SKIP)}\ \frac{}{\langle E, \mathsf{skip}\rangle \Downarrow E}$$

$$\text{(S-TRY-1)}\ \frac{\langle E, S\rangle \Downarrow E'}{\langle E, \mathsf{try}\ S\ h_1 \ldots h_n\rangle \Downarrow E'} \qquad \text{(S-TRY-2)}\ \frac{\langle E, S\rangle \Downarrow err, E' \quad \langle E', S'\rangle \Downarrow \hat{E}''}{\langle E, \mathsf{try}\ S\ \cdots \mathsf{catch}(err)\ S' \cdots\rangle \Downarrow \hat{E}''}$$

$$\text{(S-TRY-3)}\ \frac{\langle E, S\rangle \Downarrow err, E' \quad err \notin \{err_1, \ldots, err_n\}}{\langle E, \mathsf{try}\ S\ \mathsf{catch}(err_1)\ S_1 \cdots \mathsf{catch}(err_n)\ S_n\rangle \Downarrow err, E'} \qquad \text{(S-SERR-1)}\ \frac{\langle E, e\rangle \Downarrow err}{\langle E, R[e]\rangle \Downarrow err, E}$$

$$\text{(S-NEW)}\ \frac{\begin{array}{c} r = mkrec(A) \quad p = next(\eta, live(\gamma, \rho)) \quad \gamma' = \gamma[x = p] \\ \rho' = \rho[p \mapsto r[f_1 = \gamma'(x_1), \ldots, f_n = \gamma'(x_n)]] \end{array}}{\langle \gamma; \rho; \eta, x := \mathsf{new}\ A\{f_1 = x_1, \ldots, f_n = x_n\}\rangle \Downarrow \gamma'; \rho'; (p, |r|)} \qquad \text{(S-SERR-2)}\ \frac{\langle E, S_1\rangle \Downarrow err, E'}{\langle E, S_1; S_2\rangle \Downarrow err, E'}$$

Table 3. Semantics of Statements

Semantics of Expressions Table 2 defines big-step relations for expressions, $(\Downarrow) : \mathit{Env} \times \mathit{Expr} \times \widehat{\mathit{Val}}$. The semantics for variable lookup (S-VAR), integer constants (S-INT), total binary operators (S-BINOP), pointer equality (S-EQP), record field projections (S-PRJ-1), (S-PRJ-2) are entirely standard. Casting follows the behaviour of the Java bytecode *checkcast* operation in that it will always perform a runtime check to make sure that the pointer is cast to a subtype of the actual type of the record pointed to. The (S-CAST-1) rule makes use of a function to extract the runtime type of a pointer from the environment, defined as $T(p, E) = A$ iff $E(p) = \{A, \ldots\}$, i.e., if p points to a record of runtime type A.

For expressions, rules (S-PRJ-2) and (S-CAST-3) are the only sources of exceptions, originating from dereference of a nil pointer and casting to an unsupported record type.

Exception Propagation Exception propagation in expressions is achieved using the following evaluation context, which, together with the rule (S-EERR), defines how an exception can be propagated from within an expression to the top level. Let $\star$ range over the binary operators $\bullet$ and the pointer equality $=$.

$$Q \quad ::= \quad [\,] \mid Q \star e \mid e \star Q \mid (A)\ Q \mid (Q : \mathsf{int}) \mid Q.f$$

For our purposes, the evaluation context provides a way to combine all the exceptions propagation rules into one rule.

Semantics of Statements Table 3 defines a big-step operational semantics of the form, $(\Downarrow) : \mathit{Env} \times \mathit{Stmt} \times \widehat{\mathit{Env}}$. Iteration is provided by the *while* statement defined by (S-WHILE-1) and (S-WHILE-2). Depending on the iteration predicate the execution either terminates or continues with the body of the while in sequence with the while itself.

Choice is provided by the *if* statement defined by (S-IF-1) and (S-IF-2). Depending on the branch predicate either the left or the right branch is chosen for execution. Assignment to variables and fields is defined by (S-ASSV) and (S-ASSF-1) together with (S-ASSF-2) respectively. Exceptions can be caught and handled by the *try-catch* statement defined by (S-TRY-1),(S-TRY-2) and (S-TRY-3). If an exception occurs during the execution of the body (S) of the *try-catch* and there is a handler for that kind of exception control is transferred to the corresponding exception handler.

Evaluation of sequences is defined by (S-SEQ-1) and (S-SEQ-2) and, finally, evaluation of the skip statement is defined by (S-SKIP). Of the rules, (S-ASSF-2) is the only rule to introduce new exceptions; assignments to fields fail when the pointer that should point to the record is the nil pointer.

Record Allocation The only non-standard statement is record allocation. Pointer fields which have annotation $\top$ denote a *definitely non-nil field*. Since we do not allow the types of fields to change during computation it is important that such fields are not assigned a nil pointer as default

$$\text{(ST-INT)}\ \frac{\sigma_1 <: \sigma_2}{\mathsf{int}\ \sigma_1 <: \mathsf{int}\ \sigma_2} \qquad \text{(ST-VENV)}\ \frac{\forall x \in \mathit{dom}(\Gamma_2)\ .\ \Gamma_1(x) <: \Gamma_2(x)}{\Gamma_1 <: \Gamma_2} \qquad \text{(ST-RECID)}\ \frac{\Delta(A_1) <: \Delta(A_2)}{A_1 <: A_2}$$

$$\text{(ST-REC)}\ \frac{\pi_2 \subseteq \pi_1}{\pi_1 <: \pi_2} \qquad \text{(ST-PTR)}\ \frac{A_1 <: A_2 \quad \mathbf{v}_1 <: \mathbf{v}_2 \quad \mathbf{p}_1 <: \mathbf{p}_2 \quad \mathbf{o}_1 <: \mathbf{o}_2}{A_1^{\mathbf{v}_1}\ \mathbf{p}_1\ \mathbf{o}_1 <: A_2^{\mathbf{v}_2}\ \mathbf{p}_2\ \mathbf{o}_2} \qquad \text{(ST-ENV)}\ \frac{\Gamma_1 <: \Gamma_2 \quad nt_1 <: nt_2}{\Gamma_1; nt_1 <: \Gamma_2; nt_2}$$

Table 4. Subtypes

value. It is for this purpose that the *new* statement contains an initialisation specification. The *new* statement creates a new record using the $mkrec$ function, which creates a record by giving each field a default value based on the type of the field (0 for integer fields as nil for pointer fields). Thereafter, it allocates a new pointer using the *next* and *live* functions from the allocation model, and updates the fields according to the specification. This way, all fields that may not contain nil pointers are updated in a way that is atomic with respect to the view of the type system. That all fields are updated is guaranteed by the typing of the initialisation specification, see (T-NEW) below. Note that he bound variable x is available locally to allow for self-cycles.

Exception Propagation Exception propagation of exceptions originating from expressions is provided by the following context together with (S-SERR-1).

$$R \ ::= \ [\,] \mid \mathsf{while}\ R\ S \mid \mathsf{if}\ R\ S_1\ S_2$$

Propagation of errors in statements is only needed for sequencing and defined by (S-SERR-2).

4 Types

In this section we present the details of the information flow aware type system. The basic structure of top level judgments in the type system is of the form

$$\Sigma \vdash_{ct} S \Rightarrow \Sigma', \xi$$

which is read as follows: the statement S is type correct in the environment type Σ and the security context ct, yielding an environment type Σ' and an *exception type* ξ.

The fact that we have an incoming and an outgoing type environment reflects the fact that we have a *flow sensitive* type system, allowing variables to change their security level during a computation.

Now we introduce the details of these type components before discussing the actual typing rules.

Value and Environment Types The syntax of the value and environment types was introduced in Section 3. The security levels, ranged over by σ, are secret, H, and public, L. Since the security levels are used for several different purposes, for the readability we use additional meta variables ct, nt, $\mathbf{p}$ and $\mathbf{o}$ to range over the security levels representing the security context, the pointer context type, the pointer value security type and the pointer object security type respectively.

As was discussed in Section 2 we will track the domain of the pointers using annotations in the pointer types; the domain annotations are ranged over by $\mathbf{v}$, where $\top$ indicates the definite absence of nil and $\bot$ indicates the possibility of nil.

The record types, π, are partial maps from field identifiers to primitive types, τ, the variable environment types, Γ, are maps from variables to primitive types, the heap types, δ, are maps from pointers to pointer types and the record type environments, Δ, from record identifiers to record types. Finally, the environment types, $\Sigma = \Gamma; nt$, are pairs of a variable environment type and the type of the pointer context.

Exception Types An exception type ξ is a partial function from err to pairs of a security level and an environment type, written as $\sigma\ @\ \Sigma$. If a statement is typable in ξ, and $\xi(err) = \sigma\ @\ \Sigma$, then the computation may yield the exception err, encoding information of security level σ, and resulting in (i.e., being raised at) an environment with type Σ. If $err \notin \mathit{dom}(\xi)$ then the exception err is not a possible result of the statement.

Subtypes and Least Upper Bounds For security annotations we define subtyping to be the smallest transitive and reflexive relation satisfying $\sigma <: \mathsf{H}$. Similarly, for pointer domain annotations $\top <: \bot$. Value subtyping is defined structurally by covariance, with record subtyping limited to *width* subtyping[9], so that $\pi_1 <: \pi_2$ iff $\pi_2 \subseteq \pi_1$. The rules are presented below: Like most systems for the information flow analysis of objects, but in contrast to [1], the types of records are flow *in*sensitive. This, together with our use of width subtyping [9], avoids problems with pointer aliasing since it guarantees that all pointers pointing to the same record must agree on the types of the common fields. The rules for the subtypes are defined in Table 4.

The least upper bound of two types T_1 and T_2 is defined

$$\text{(T-PRJ-1)}\ \frac{\Sigma \vdash_{ct} e : A^{\top}\ \mathbf{p}\ \mathbf{o}, \xi \qquad (f : \tau) \in \Delta(A)}{\Sigma \vdash_{ct} e.f : \tau^{\mathbf{o}}, \xi} \qquad \text{(T-PRJ-2)}\ \frac{\Sigma \vdash_{ct} e : A^{\bot}\ \mathbf{p}\ \mathbf{o}, \xi \qquad (f : \tau) \in \Delta(A)}{\Sigma \vdash_{ct} e.f : \tau^{\mathbf{o}}, \xi \sqcup \{\mathsf{np} \mapsto ct \sqcup \mathbf{o}\ @\ \Sigma\}}$$

$$\text{(T-CAST)}\ \frac{\Sigma \vdash_{ct} e : A_2^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o}, \xi}{\Sigma \vdash_{ct} (A_1)\ e : A_1^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o}, \xi \sqcup \{\mathsf{cc} \mapsto \mathbf{o}\ @\ \Sigma\}} \qquad \text{(T-INT)}\ \frac{}{\Sigma \vdash_{ct} n : \mathsf{int}\ \mathsf{L}, \emptyset} \qquad \text{(T-NIL)}\ \frac{}{\Sigma \vdash_{ct} \mathsf{nil} : A^{\bot}\ \mathsf{L}\ \mathsf{L}, \emptyset}$$

$$\text{(T-COERCE)}\ \frac{\Sigma \vdash_{ct} e : A^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o}, \xi}{\Sigma \vdash_{ct} (e : \mathsf{int}) : \mathsf{int}\ \mathbf{p}, \xi} \qquad \text{(T-EQ-P)}\ \frac{\Sigma \vdash_{ct} e_1 : A_1^{\mathbf{v}_1}\ \mathbf{p}_1\ \mathbf{o}_1, \xi_1 \qquad \Sigma \vdash_{ct} e_2 : A_2^{\mathbf{v}_2}\ \mathbf{p}_2\ \mathbf{o}_2, \xi_2}{\Sigma \vdash_{ct} e_1 = e_2 : \mathsf{int}\ (\mathbf{o}_1 \sqcup \mathbf{o}_2), \xi_1 \sqcup \xi_2}$$

$$\text{(T-VAR)}\ \frac{\Sigma(x) = \tau}{\Sigma \vdash_{ct} x : \tau, \emptyset} \qquad \text{(T-BINOP)}\ \frac{\Sigma \vdash_{ct} e_1 : \mathsf{int}\ \sigma_1, \xi_1 \qquad \Sigma \vdash_{ct} e_2 : \mathsf{int}\ \sigma_2, \xi_2}{\Sigma \vdash_{ct} e_1 \bullet e_2 : \mathsf{int}\ (\sigma_1 \sqcup \sigma_2), \xi_1 \sqcup \xi_2}$$

Table 5. Expression Type Rules

as the smallest value T satisfying $T_1 <: T \wedge T_2 <: T$. Least upper bounds for exception types is defined as the union of the two exception types while merging pairs associated with the same exception using $\sigma_1\ @\ \Sigma_1 \sqcup \sigma_2\ @\ \Sigma_2 = \sigma_1 \sqcup \sigma_2\ @\ \Sigma_1 \sqcup \Sigma_2$.

Expression Type Rules The expression type rules are of the form $\Sigma \vdash_{ct} e : \tau, \xi$, which is read as follows: the expression e is type correct in the environment type Σ in the security context ct, yielding either a value of type τ or an exception defined by ξ.

We use a global record type environment $\Delta \in RecID \rightarrow RecType$ to record the record types associated with the record identifiers.

There are two type rules for field projection: (T-PRJ-1) and (T-PRJ-2). The former rule corresponds to the case where the possibility of a nil-pointer exception can be ruled out by the domain annotation. The field projection expression is able to fetch values from a record even if the value of the pointer is secret, without having to consider the result as a secret. This possibility comes from the separation between the security level of the pointer value and the security level of the record pointed to. The object security type acts as an override for the security types of the record type. Fetching data from a secret record returns secret results enforced by the following function:

$$(\mathsf{int}\ \sigma_1)^{\sigma_2} = \mathsf{int}\ (\sigma_1 \sqcup \sigma_2)$$
$$(A^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o})^{\sigma} = A^{\mathbf{v}}\ (\mathbf{p} \sqcup \sigma)\ (\mathbf{o} \sqcup \sigma))$$

Subtyping allows us to disregard fields in a record pointed to by a pointer, by changing the type of the pointer to a smaller type. Casting, (T-CAST), is intended to reverse this operation by upgrading a pointer type to a wider pointer type. The correctness of such an operation is not statically decidable, which is why a runtime type check has to be performed. If this type check fails, a class cast exception is thrown as documented by exception type. The integers introduce public values, (T-INT). The same holds for the (syntactic) nil pointer, (T-NIL). The coercion expression, (T-COERCE), models the failure of pointer opaqueness by lifting pointer values to integers. The result of the function depends only on the values of the pointer, and safely ignores the security annotations associated with the referred record. The binary operators, (T-BINOP), are total and are assumed not to cause abnormal termination, and simply return a result which is as secret as the most secret of its subexpressions. Note that the result of comparing two pointer expressions for equality, (T-EQ-P), is independent of the security level of the pointer value. This is not surprising, since equality in the presence of only nil pointer constants is an opaque operation.

Statement Type Rules As mentioned previously, the statement types are of the form $\Sigma \vdash_{ct} S \Rightarrow \Sigma', \xi$, which is read as follows: the statement S is type correct in the environment type Σ and security context ct, yielding either an environment of type Σ' or an exception defined by ξ. The statement type rules are found in Table 6.

Allocation, (T-NEW), allocates a new record and a pointer and updates the pointer context. Allocation has the property that a well-typed *new* will produce well-formed records – even for *non-nil recursive record types*. Just like any other update, updating the pointer context (i.e., allocating) in a secret context causes the pointer context to become secret. We assume that allocation does not fail, which means that we assume that the memory is infinite.

All of the *while* statement, (T-WHILE), the if-statement, (T-IF), and the exceptions may introduce indirect leaks. Any differences in low modifications in the body of an *if* or a *while* encodes information about the guarding expression.

Exceptions introduce conditional branches to the associated exception handler — any modifications done after an instruction that may cause exceptions may encode information about the parameters causing the exception. Consider

$$\text{(T-WHILE)}\ \frac{\Sigma' \vdash_{\sigma'} e : \mathsf{int}\ \sigma, \xi_1 \qquad ct \sqcup \sigma \sqcup lvl(\xi_1) = \sigma' \qquad \Sigma' \vdash_{\sigma'} S \Rightarrow \Sigma'', \xi_2 \qquad \Sigma'' <: \Sigma' \qquad \Sigma <: \Sigma'}{\Sigma \vdash_{ct} \mathsf{while}\ e\ S \Rightarrow \Sigma', \xi_1 \sqcup \xi_2}$$

$$\text{(T-ASSV)}\ \frac{\Sigma \vdash_{ct} e : \tau, \xi \qquad lvl(\xi) = \sigma \qquad cmp(\Sigma'(x), \tau)}{\Sigma \vdash_{ct} x := e \Rightarrow \Sigma'[x : \tau^{ct \sqcup \sigma}], \xi}$$

$$\text{(T-IF)}\ \frac{\Sigma \vdash_{ct} e : \mathsf{int}\ \sigma, \xi \qquad ct \sqcup \sigma \sqcup lvl(\xi) = \sigma' \qquad \Sigma \vdash_{\sigma'} S_i \Rightarrow \Sigma_i, \xi_i \qquad i \in \{1, 2\}}{\Sigma \vdash_{ct} \mathsf{if}\ e\ S_1\ S_2 \Rightarrow \Sigma_1 \sqcup \Sigma_2, \xi \sqcup \xi_1 \sqcup \xi_2}$$

$$\text{(T-SEQ)}\ \frac{\Sigma \vdash_{ct} S_1 \Rightarrow \Sigma', \xi_1 \qquad lvl(\xi_1) = \sigma \qquad \Sigma' \vdash_{ct \sqcup \sigma} S_2 \Rightarrow \Sigma'', \xi_2}{\Sigma \vdash_{ct} S_1; S_2 \Rightarrow \Sigma'', \xi_1 \sqcup \xi_2}$$

$$\text{(T-ASSF-1)}\ \frac{\Sigma \vdash_{ct} e : \tau, \xi \qquad \Sigma'(x) = A^{\top}\ \mathbf{p}\ \mathbf{o} \qquad (f : \tau') \in \Delta(A) \qquad lvl(\xi) = \sigma \qquad \tau^{ct \sqcup \mathbf{o} \sqcup \sigma} <: \tau'}{\Sigma \vdash_{ct} x := e \Rightarrow \Sigma', \xi}$$

$$\text{(T-ASSF-2)}\ \frac{\Sigma \vdash_{ct} e : \tau, \xi \qquad \Sigma'(x) = A^{\bot}\ \mathbf{p}\ \mathbf{o} \qquad (f : \tau') \in \Delta(A) \qquad lvl(\xi) = \sigma \qquad \tau^{ct \sqcup \mathbf{o} \sqcup \sigma} <: \tau'}{\Sigma \vdash_{ct} x := e \Rightarrow \Sigma', \xi \sqcup \{\mathsf{np} \mapsto ct \sqcup \mathbf{o}\ @\ \Sigma'\}}$$

$$\text{(T-TRY)}\ \frac{\Sigma \vdash_{ct} S \Rightarrow \Sigma', \{err_1 \mapsto \sigma_1\ @\ \Sigma_1, \ldots, err_n \mapsto \sigma_n\ @\ \Sigma_n\} \cup \xi \qquad err_i \notin \mathit{dom}(\xi) \qquad \Sigma_i \vdash_{ct \sqcup \sigma_i} S_i \Rightarrow \Sigma_i', \xi_i \qquad i \in \{1..n\}}{\Sigma \vdash_{ct} \mathsf{try}\ S\ \mathsf{catch}(err_1)\ S_1 \ldots \mathsf{catch}(err_n)\ S_n \Rightarrow \bigsqcup_{i \in \{1..n\}} \Sigma_i' \sqcup \Sigma', \bigsqcup_{i \in \{1..n\}} \xi_i \sqcup \xi}$$

$$\text{(T-NEW)}\ \frac{\Delta(A_1) = \{f_1 : \tau_1, \ldots, f_n : \tau_n, \ldots\} \qquad f_i : A_2^{\top}\ \mathbf{p}\ \mathbf{o} \Longrightarrow i \leq n \qquad \Gamma' = \Gamma[x : A_1^{\top}\ (nt \sqcup ct)\ ct] \qquad \Gamma'(x_i) <: \tau_i \quad i \in \{1..n\} \qquad cmp(\Gamma(x), \tau)}{\Gamma; nt \vdash_{ct} x := \mathsf{new}\ A_1\{f_1 = x_1, \ldots, f_n = x_n\} \Rightarrow \Gamma'; nt \sqcup ct, \emptyset}$$

Table 6. Statement Type Rules

the following program,

```
h2 := nil; if (h1) then h2 := new A{};
  try { h2.f := 1; l := 1 } catch (np) { l := 0 }
```

where $l = 1$ if $h1 \neq 0$ and $l = 0$ if $h1 = 0$. This is where the context annotation, ct, is used – every statement (and thus expressions) has to be typable in the security context of the controlling expression or exception.

In the statement sequence instruction this is expressed by typing the second statement in the context of the exception level $lvl(\xi)$ of the first, where $lvl(\xi) = \bigsqcup\{\sigma \mid \xi(err) = (\sigma, \Sigma), err \in \mathit{dom}(\xi)\}$. In the *if* and the *while* not only the security level of the guarding expression but also the exception level decides the context of their bodies.

However, there are more places where we have to consider the effect of exceptions namely in the instructions that modify the environment, i.e., variable and field updates. If an exception occurs in the expression providing the value for the update this will prevent the update from occurring. For this reason the context of the actual write has to be at least as secret as the exception level of the expression.

As is the case with the field projection above, the type rule for field update, (T-ASSF), gives us the possibility to use low parts of records even after secret allocation.

From the variable assignment rule (T-ASSV) we can see that assigning a value to a variable may cause the the security type of that variable to change (i.e., we have a *flow sensitive* type system). However, as is standard we do not allow it to change the underlying type of the variable. As seen in the type rule for variable assignment, (T-ASSV), this is enforced by the demand of type compatibility, expressed by the predicate $cmp(\cdot)$, defined as follows:

$$cmp(\mathsf{int}\ \sigma_1, \mathsf{int}\ \sigma_2)$$
$$cmp(A^{\mathbf{v}_1}\ \mathbf{p}_1\ \mathbf{o}_1, A^{\mathbf{v}_2}\ \mathbf{p}_2\ \mathbf{o}_2)$$

Since the (security) types of variables may change on assignment, the sequencing rule, (T-SEQ), demands that the second statement is correct in the type environment resulting from the first statement.

Furthermore, the flow sensitivity is what forces us to tag the exception types with the environment type in which the expression was thrown. As is seen in the type rule for the *try-catch* (T-TRY), this annotation is used to make sure that we can safely transfer control from the location of the exception to the corresponding exception handler by demanding that the exception handler is typable in the environment type carried by the exception type. To understand how the *try-catch* statement prevents succeeding statements from executing in the context of caught exceptions note how those are removed from the exception type of the entire *try-catch*, which will prevent them from being propagated to the security context of the succeeding statement (if any).

Consider the following program where S denotes any statement and h is a pointer that could be nil.

```
try { h.f := 0; } catch (np) { skip; }; S
```

$$\text{(LE-INT-L)}\ \frac{}{\beta \vdash n \sim_{\mathsf{int\ L}} n} \qquad \text{(LE-INT-H)}\ \frac{}{\beta \vdash n_1 \sim_{\mathsf{int\ H}} n_2} \qquad \text{(LE-PTR-PL)}\ \frac{}{\beta \vdash p \sim_{P_{\mathsf{L}}} p}$$

$$\text{(LE-PTR-PH)}\ \frac{}{\beta \vdash p_1 \sim_{P_{\mathsf{H}}} p_1} \qquad \text{(LE-PTR-OL)}\ \frac{p_i \neq \mathsf{nil} \quad (p_1, p_2) \in \beta}{\beta \vdash p_1 \sim_{O_{\mathsf{L}}} p_2} \qquad \text{(LE-PTR-OL)}\ \frac{}{\beta \vdash \mathsf{nil} \sim_{O_{\mathsf{L}}} \mathsf{nil}}$$

$$\text{(LE-PTR-OH)}\ \frac{}{\beta \vdash p_1 \sim_{O_{\mathsf{H}}} p_2} \qquad \text{(LE-PTR)}\ \frac{\beta \vdash \rho_1 \sim_{P_{\mathbf{p}}} p_2 \quad \beta \vdash p_1 \sim_{O_{\mathbf{o}}} p_2}{\beta \vdash p_1 \sim_{A^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o}} p_2}$$

$$\text{(LE-REC)}\ \frac{\forall (f : \tau) \in \pi\ .\ \beta \vdash r_1.f \sim_{\tau} r_2.f}{\beta \vdash r_1 \sim_{\pi} r_2} \qquad \text{(LE-VENV)}\ \frac{\forall x \in \mathit{dom}(\Gamma)\ .\ \beta \vdash \gamma_1(x) \sim_{\Gamma(x)} \gamma_2(x)}{\beta \vdash \gamma_1 \sim_{\Gamma} \gamma_2}$$

$$\text{(LE-HEAP)}\ \frac{\forall (p_1, p_2) \in \beta\ .\ T(p_1, \rho_1) = A = T(p_2, \rho_2) \quad \beta \vdash \rho_1(p_1) \sim_{\Delta(A)} \rho_2(p_2)}{\beta \vdash \rho_1 \sim \rho_2} \qquad \text{(LE-ENV)}\ \frac{\beta \vdash \gamma_1 \sim_{\Gamma} \gamma_1 \quad \beta \vdash \rho_1 \sim \rho_2 \quad \vdash \eta_1 \sim_{nt} \eta_2}{\beta \vdash \gamma_1; \rho_1; \eta_1 \sim_{\Gamma;nt} \gamma_2; \rho_2; \eta_2}$$

Table 7. Low-Equivalence of Values

Since the *try-catch* is catching nil-pointer exceptions the nil-pointer exception is removed from the exception type of the *try-catch* statement, which (in this case) results in the empty exception type, the context of S will not be touched by the sequence statement. Without the *try-catch*, S would execute in the contexts of the nil-pointer exception i.e., the security level of h.

5 Noninterference

The security condition that the type system aims to guarantee is phrased as a *noninterference* property. Noninterference is the prevailing formalization of absence of information leaks. A program is secure – *noninterfering* — if whenever the program is run in environments that are indistinguishable to the attacker, the results of running the program are also indistinguishable. The key to the definition is to define the notion of "indistinguishable" in an appropriate way. Typically, indistinguishability is defined with respect to a classification of the different parts of the environments into secret and public information — the environment type. Given that the attacker can inspect only the public parts of the environment, two environments are indistinguishable to the attacker if their public parts are equal. Hence the frequently used name *low-equivalence* for the indistinguishability relation.

Low-equivalence Informally, two values are low-equivalent with respect to a security type, if all their public parts are low-equivalent with respect to their respective security types. In order to define this notion we need to traverse the heap. In a setting where pointers are opaque we do not need to insist that pointer values are identical – it is sufficient that there is a bijective renaming that relates them. The use of a bijection (on the low-reachable sub-domains of the heaps) in the formulation of low-equivalence was pioneered by Banerjee and Naumann[3]. The bijection has two purposes: firstly, it makes the low-equivalence relation inductively definable in the presence of cycles on the heap and, secondly, it has the beneficial side effect of allowing equality comparisons of certain pointers with secret values in addition to the ones with public values. Since the work by Banerjee and Naumann abstracts away from the values of the pointers, parameterizing the low-equivalence relation is not strictly necessary – it would suffice to apply a renaming before relating two values. In the present work the bijection plays a more crucial role, since the actual pointer values *are* exposed, which prohibits us from renaming values.

Let β be a bijection on some subset of Ptr. Table 7 defines the meaning of the security types for values, *low-equivalence*, as a family of partial equivalence relations indexed over the family of heap bijections and the value types.

Two public integers are low-equivalent if they are the same integer (LE-INT-L). Any two secret integers are low-equivalent (LE-INT-H), reflecting the fact that any two secret integers look the same to the attacker. Pointers have more than one security annotation. As with integers there is a security level for the value of the pointers. Thus, two (value-wise) public pointers are low-equivalent if they have the same value (LE-PTR-L) and any two (value-wise) secret pointers are low-equivalent (LE-PTR-H). The remaining security annotations for pointers deals with *pointer related object properties*. Two pointers are related with respect to O_L if the records pointed to in the respective heap have the same runtime type and the records are low-equivalent field for field (LE-PTR-OL). The reason for demanding that the records have the same runtime type is because the cast operation can be used to distinguish between records of different

runtime types. Pointers typed O_H pose no demands on the records pointed to (LE-PTR-OH). Similar to above, demands of low-equivalence for records are carried by the bijection to the low-equivalence rule for the heap in which they are enforced. Two heaps are low-equivalent with respect to the bijection if all records pointed to by pointers in the bijection are low-equivalent with respect to the record type.

Low-equivalence for pointers with respect to the entire pointer type demands low-equivalence with respect to the value and object security levels (LE-PTR). Records, variable environments and environments are then related by pointwise extension (LE-REC, LE-VENV, LE-ENV).

Finally, an important property of $\sim_\Sigma$ is that it is a *partial equivalence relation* (PER) – i.e., that it is transitive and symmetric.

Lemma 5.1. *(LE-ENV) is a partial equivalence relation., i.e.,* $\beta \vdash E_1 \sim_\Sigma E_2 \implies \beta^{op} \vdash E_1 \sim_\Sigma E_2$ *and* $\beta_1 \vdash E_1 \sim_\Sigma E_2 \wedge \beta_2 \vdash E_2 \sim_\Sigma E_3 \implies \beta_1 \circ \beta_2 \vdash E_1 \sim_\Sigma E_3$

Proof. By induction on equivalence derivation. □

Low-equivalence with respect to Exceptions We define the family of low-equivalence relations on $\widehat{Val}$ as the smallest family of symmetric relations satisfying:

$$\frac{\beta \vdash v_1 \sim_\tau v_2}{\beta \vdash v_1 \sim_{\xi,\tau} v_2} \qquad \frac{\xi(err) = \mathsf{H} \mathbin{@} \Sigma'}{\beta \vdash err \sim_{\xi,\tau} v}$$

$$\frac{\begin{array}{c} err_1 \in \mathit{dom}(\xi) \qquad err_2 \in \mathit{dom}(\xi) \\ \xi(err_1) : \mathsf{L} \mathbin{@} \Sigma_1 \wedge \xi(err_2) : \mathsf{L} \mathbin{@} \Sigma_2 \implies err_1 = err_2 \end{array}}{\beta \vdash err_1 \sim_{\xi,\tau} err_2}$$

Similarly for $\widehat{Env}$:

$$\frac{\beta \vdash E_1 \sim_\Sigma E_2}{\beta \vdash E_1 \sim_{\xi,\Sigma} E_2} \qquad \frac{\xi(err) = \mathsf{H} \mathbin{@} \Sigma' \quad \beta \vdash E_1 \sim_\Sigma E_2}{\beta \vdash E_1 \sim_{\xi,\Sigma} err, E_2}$$

$$\frac{\begin{array}{c} err_1 \in \mathit{dom}(\xi) \qquad err_2 \in \mathit{dom}(\xi) \\ \xi(err_1) : \mathsf{L} \mathbin{@} \Sigma_1 \wedge \xi(err_2) : \mathsf{L} \mathbin{@} \Sigma_2 \implies err_1 = err_2 \\ \beta \vdash E_1 \sim_{\Sigma_1 \sqcup \Sigma_2} E_2 \end{array}}{\beta \vdash err_1, E_1 \sim_{\xi,\Sigma} err_2, E_2}$$

We note that the resulting relation for environments is not transitive. However, it is transitive on the distinct domain of environments, which is what is needed in the proof of correctness.

5.1 Soundness

Now, given the notion of low-equivalence, we formulate the notion of security, *exception-sensitive noninterference*, which is a termination-insensitive noninterference, where

$$\frac{}{\delta \vdash n : \mathsf{int}\ \sigma} \qquad \frac{}{\delta \vdash \mathsf{nil} : A^\perp\ \mathbf{p}\ \mathbf{o}}$$

$$\frac{\delta(p) <: A \qquad p \neq \mathsf{nil}}{\delta \vdash p : A^{\mathbf{v}}\ \mathbf{p}\ \mathbf{o}}$$

$$\frac{\Delta(A) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \qquad \delta \vdash v_i : \tau_i}{\delta \vdash \{A, f_1 = v_1, \ldots, f_n = v_n\} : A}$$

$$\frac{\forall x \in \mathit{dom}(\Gamma)\ .\ \delta \vdash \gamma(x) : \Gamma(x)}{\delta \vdash \gamma : \Gamma}$$

$$\frac{\forall p : A \in \delta\ .\ \delta \vdash \rho(p) : A}{\delta \vdash \rho} \qquad \frac{\delta \vdash \rho \quad \delta \vdash \gamma : \Gamma}{\delta \vdash \gamma; \rho; \eta : \Gamma; nt}$$

$$\frac{\delta \vdash v : \tau}{\delta \vdash v : \xi, \tau} \qquad \frac{\xi(err) = \sigma \mathbin{@} \Sigma}{\delta \vdash err : \xi, \tau}$$

$$\frac{\delta \vdash E : \Sigma}{\delta \vdash E : \xi, \Sigma} \qquad \frac{\xi(err) = \sigma \mathbin{@} \Sigma' \quad \delta \vdash E : \Sigma'}{\delta \vdash err, E : \xi, \Sigma}$$

Table 8. Well Formed Values

abnormal termination and the cause of the abnormal termination is considered observable but not non-termination. Specifically, this means that we cannot consider a program that causes secret exceptions as secure. Depending on what is considered the result of running the program, we get different formulations of indistinguishability. The least we must demand is that publicly observable actions are equal.

However, a formulation of noninterference that only considers public observations, is frequently not compositional[4] and, thus, hard to prove directly from a compositional type system. The solution is to find a stronger formulation of noninterference, which is compositional and a safe approximation of the original formulation, typically by extending the equality demand to the non-observable public parts, and to show that well-typedness implies the stronger relation. Since our language is not equipped with any constructions for communication with the outside world we assume an execution model where the public (low) parts of the final environment as well as the termination status are observable to the attacker. From the compositionality argument above it should be clear that this does not impose any demands that are not needed by the proof.

Assume that S is well-typed, i.e., $\Sigma \vdash_{ct} S \Rightarrow \Sigma', \xi$. To formulate a noninterference property applicable to such a judgment we need to start computing S in two low equivalent environments with respect to Σ.

To do this we will need a simple *well-formedness* relation for type environments. The rules for *well formed values* (including environments) are found in Table 8.

This makes the basic connection between *values* and

[4] since we do not have any information about the non-observable public values of the environment

types. To make the family of well-formedness relations inductively definable in the presence of cycles on the heap they are parameterized over a heap type δ, which maps all *reachable* pointers to the *runtime type* of the record pointed to.

A program is secure with respect to some initial and final type environments Σ and Σ' and the exception type ξ, if whenever the program is run on environments that are indistinguishable to the attacker, the results (modulo nontermination) are also indistinguishable. Clearly, since it is assumed that the attacker can distinguish between normal and abnormal termination we cannot allow secret exceptions propagating to the top level. Thus, the following notion of non-interference is defined only for exception types ξ s.t. $lvl(\xi) = \mathsf{L}$.

Definition 5.1 (Noninterference). *For* $lvl(\xi) = \mathsf{L}$

$$NI_{\Sigma,\Sigma',\xi}(S) \stackrel{def}{=} \forall E_1, E_2 \,.\, E_1 : \Sigma \wedge E_2 : \Sigma \wedge E_1 \sim_\Sigma E_2 \\ \wedge \langle E_1, S\rangle \to \hat{E}_1' \wedge \langle E_2, S\rangle \to \hat{E}_2' \Longrightarrow \hat{E}_1' \sim_{\Sigma',\xi} \hat{E}_2'$$

With this we can formulate the main theorem of the paper: that well-typed programs are noninterfering. A noninterference proof is essentially a preservation proof. We are proving that execution *preserves* a type invariant. Because of the form of the noninterference definition, the proof is a merge between two proofs: one proof that proves *ordinary* preservation of types, i.e., that *well-formedness* is preserved, together with one proof that proves that low-equivalence is preserved. The reason for this is that the well-formedness properties are needed in some cases in the proof of preservation of low-equivalence.

Theorem 5.1. *If* $\Sigma \vdash_{ct} S \Rightarrow \Sigma', \xi$ *and* $lvl(\xi) = \mathsf{L}$ *then* $NI_{\Sigma,\Sigma',\xi}(S)$

Proof. By induction on the type derivation. The proof is omitted for space reasons.

□

6 Future Work: Security by Transformation

The main topic of this paper has been how to deal with the indirect information leaks arising from allocation in secret contexts, with the perspective that the allocation model is fixed and deterministic.

Another way of avoiding the covert channels caused by non-opaque pointers is a *strong separation* of the heap into a *public* heap and a *secret* heap, for allocations in public and secret contexts respectively – in a style typical of "classical" military message passing systems.

In this section we present an idea on how to achieve the same effect as a strongly separated heap within a single heap system, with a combination of dynamically allocation of new identifiers for public data which can be used to form a safe coercion function, together with a type directed transformation that transforms a program free from leaks other than secret allocation to a program that is noninterfering.

The point of this transformation is that in a reasonable language, this map can be expressed in the language *without* extending the semantics. Consider the following example:

original	transformed
int L x;	**int** L x;
p1 := **new** A;	p1 := **newL** A;
if (secret)	**if** (secret)
{ p2 := **new** A{}; }	{ p2 := **new** A{}; }
p3 := **new** B;	p3 := **newL** B;
x := (p3 : **int**)	x := (p3 :L **int**)

The left program would not be type correct (nor safe) but the transformed program on the right would. In addition to allocating a new object of type A, $p1 = \mathsf{newL}\ A$ also allocates a new unique identifier and associates the newly allocated pointer with this identifier. Again, both a pointer and an identifier is allocated by $p3 = \mathsf{newL}\ B$ and it is this identifier, *not* the pointer representation, that is returned by the $x = (p3 :\mathsf{L}\ int)$ instruction at the end of the program. In contrast to the pointer representation of $p3$ the identifier associated with the pointer is not affected by the allocation in the secret context, since that allocation does not allocate an identifier.

The correctness of such a transformation relies on that "reasonable" programs using non-opaque pointers are not dependent on a *particular* allocation model, i.e., their semantics is independent of the values of the allocated pointers. With this view one could see such a transformation as a refinement of the original program.

7 Conclusion

This paper has presented the problem of information leakage in the presence of non-opaque pointers, and presented a type-based analysis for a simple imperative language which tracks the use of opaque operations in order to eliminate a class of information flows not previously modeled by either theoretical or practical systems. The type system combines a number of features, including separate types for pointer value and record pointed to and value-flow information about the initialisation status of pointers. On the semantic side, we adopted an abstract and rather general model of allocation to represent many possible implementations of non-opaque pointers, and were able to prove a noninterference result for the type system, demonstrating that key reasoning methods for opaque pointers can still be applied in a non-opaque setting.

Acknowledgements Thanks to Niklas Broberg, Tobias Gedell, Ulf Norell and Andrei Sabelfeld for helpful comments and feedback. Thanks to the anonymous referees for numerous helpful comments and suggestions. This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the Thirty-third Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 91–102, January 2006.

[2] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

[3] Anindya Banerjee and David Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, pages 131–177, sep 2005.

[4] Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[5] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[6] René Rydhof Hansen. *Flow Logic for Language-Based Safety and Security*. PhD thesis, Technical University of Denmark, 2005.

[7] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. In Fausto Spoto, editor, *BYTECODE'05*, ENTCS. Elsevier, April 2005.

[8] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[9] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

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

[11] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.