Security properties of typed applets

Xavier Leroy, François Rouaix

To cite this version:

Xavier Leroy, François Rouaix. Security properties of typed applets. POPL 1998: 25th symposium Principles of Programming Languages, Jan 1998, San Diego, United States. ACM, pp.391-403, <10.1145/268946.268979>. <hal-01499963>

HAL Id: hal-01499963
https://hal.inria.fr/hal-01499963

Submitted on 1 Apr 2017

Security properties of typed applets

Xavier Leroy François Rouaix
INRIA Rocquencourt[*]

Abstract

This paper formalizes the folklore result that strongly-typed applets are more secure than untyped ones. We formulate and prove several security properties that all well-typed applets possess, and identify sufficient conditions for the applet execution environment to be safe, such as procedural encapsulation, type abstraction, and systematic type-based placement of run-time checks. These results are a first step towards formal techniques for developing and validating safe execution environments for applets.

1 Introduction

What, exactly, makes strongly-typed applets more secure than untyped ones? Most frameworks proposed so far for safe local execution of foreign code rely on strong typing, either statically checked at the client side [13, 34], statically checked at the server side and cryptographically signed [28], or dynamically checked by the client [4]. However, the main property guaranteed by strong typing is type soundness: "well-typed programs do not go wrong", e.g. do not apply an integer as if it were a function. While violations of type soundness constitute real security threats (casting a well-chosen string to a function or object type allows arbitrary code to be executed), there are many more security concerns, such as integrity of the running site (an applet should not delete or modify arbitrary files) and confidentiality of user's data (an applet should not divulge personal information over the network). The corresponding security violations do not generally invalidate type soundness in the conventional sense.

If we examine the various security problems identified for Java applets [8], only few of them cause a violation of Java type soundness [15]; the others correspond to malicious, but well-typed, uses of improperly protected functions from the applet's execution environment [3]. Another typical example is the ActiveX applet doing a Trojan attack on the Quicken home-banking software described in [6]: money gets transferred from the user's bank account to some offshore account, all in a perfectly type-safe way.

On these examples, it is intuitively obvious that security properties must be enforced by the applet's execution environment. It is the environment that eventually decides which computer resources the applet can access. This is the essence of the so-called "sandbox model". Strong typing comes in the picture only to guarantee that this environment is used in accordance with its publicized interface. For instance, typing prevents an applet from jumping in the middle of the code for an environment function, or scanning the whole memory space of the browser, which would allow the applet to abuse or bypass entirely the execution environment.

The purpose of this paper is to give a formal foundation to the intuition above. We formulate and prove several security properties that all well-typed applets possess. Along the way, we identify sufficient conditions for the execution environment to be safe, such as procedural encapsulation, type abstraction, and systematic type-based placement of run-time checks. These results are a first step towards formal techniques for developing and validating safe execution environments for applets.

The remainder of this paper is organized as follows. Section 2 introduces a simple language for applets and their environment, and formalizes the security policy that they must obey. Section 3 proves a security property based on the notion of lexical scoping, then extends it to take procedural abstraction into account. In section 4, we equip our language with a simple type system, which is used in section 5 to prove three type-based security properties, two relying on run-time checks and the other on a combination of run-time checks and type abstraction. After a brief parallel with object-oriented languages in section 6, related work is discussed in section 7, followed by concluding remarks in section 8.

2 The language and its security policy

2.1 The language

The language we consider in this paper is a simple lambda-calculus with base types (integers, strings, ...), pairs as the only data structure, and references in the style of ML. The syntax of terms, with typical element a, is as follows:

Terms:

$a ::= x$	identifiers
$\mid b$	constant of base type (integer, ...)
$\mid \lambda x.a$	function abstraction
$\mid a_1(a_2)$	function application
$\mid (a_1, a_2)$	pair construction
$\mid \mathtt{fst}(a) \mid \mathtt{snd}(a)$	pair projections
$\mid \mathtt{ref}(a)$	reference creation
$\mid\ !a$	dereferencing
$\mid a_1 := a_2$	reference assignment

References are included in the language from the beginning not only to account for imperative programming (all kinds

To appear in the 25th ACM conference on Principles of Programming Languages, January 1998.

[*] Authors' address: INRIA Rocquencourt, projet Cristal, B.P. 105, 78153 Le Chesnay, France. E-mail: Xavier.Leroy@inria.fr, Francois.Rouaix@inria.fr. This work has been partially supported by GIE Dyade under the "Verified Internet Protocols" project.

of assignment on variables and mutable data structures such as arrays can easily be modeled with references), but also to provide easily observable criteria on which we base the security policy.

2.2 The security policy

The security policy we apply to applets is based on the notion of sensitive store locations: locations of references that an applet must not modify during its execution, or more generally references that can be modified during the applet execution, but whose successive contents must always satisfy some invariant, i.e. remain within a given set of permitted values.

The first motivation for this policy is to formalize the intuitive idea that an applet must not trash the memory of the computer executing it. In particular, the internal state of the browser, the operating system, and other applications running on the machine must not be adversely affected by the applet.

This security policy can also be stretched to account for input/output behavior, notably accesses to files and simple cases of network connections. A low-level, hardware-oriented view of I/O is to consider hardware devices such as the disk controller and network interface as special locations in the store; I/O is then controlled by restricting what can be written to these locations. For a higher-level view, each file or network connection can be viewed as a reference, which can then be controlled independently of others. Here, the file system, the name service and the routing tables become dictionary-like data structures mapping file names and network addresses to the references representing files and connections.

(By concentrating on writes to sensitive locations, we focus on integrity properties of the system running the applet. It is also possible to control reads from sensitive locations, thus establishing privacy properties. We will not do it in this paper for the sake of simplicity, but the results of section 3 also extend to controlled reads.)

2.3 The instrumented semantics

To enforce the security policy, we give a semantics to our language that monitors reference assignments, and reports run-time errors in the case of illegal writes. We use a standard big-step operational semantics in the style of [25, 30, 18]. Source terms are mapped to values, which are terms with the following syntax:

Values:	
$v ::= b$	values of base types
$\mid \lambda x.a[e]$	function closures
$\mid (v_1, v_2)$	pairs of values
$\mid \ell$	store locations
Results:	
$r ::= v/s$	normal termination
$\mid$ `err`	write violation detected
Environments:	
$e ::= [x_1 \leftarrow v_1 \ldots, x_n \leftarrow v_n]$	
Stores:	
$s ::= [\ell_1 \leftarrow v_1 \ldots, \ell_n \leftarrow v_n]$	

The evaluation relation, defined by the inference rules in figure 1, is written $\varphi, e, s \vdash a \rightarrow r$, meaning that in evaluation environment e, initial store s, and store control φ, the source term a evaluates to the result r, which is either `err` if an illegal write was detected or a pair v/s' of a value v for the source term and a modified store s'.

The only unusual ingredient in this semantics is the φ component, which maps store locations to sets of values: if $\varphi(\ell)$ is defined, values written to the location ℓ must belong to the set $\varphi(\ell)$, otherwise a run-time error `err` is generated (and propagated by rules 12–21); if $\varphi(\ell)$ is undefined, any value can be stored at ℓ. (See rules 10 and 11.) For instance, taking $\varphi(\ell) = \emptyset$ prevents any assignment to ℓ.

The rules for propagating the `err` result and aborting execution (rules 12–21) are the same rules as for propagating run-time type errors (`wrong`) in [30]; the only difference is that we have no rules to detect run-time type errors, thus making no difference between run-time type violations and non-terminating programs (no derivations exist in both cases): the standard type soundness theorems show that type violations cannot occur at run-time in well-typed source terms.

An unusual aspect of our formalism is that the store control φ must be given at the start of the execution. The reason is that, with big-step operational semantics, it does not suffice to perform a regular evaluation $e, s \vdash a \rightarrow v/s'$ and observe the differences between s and s' to detect illegal writes. For one thing, we would not observe temporary assignments, where a malicious applet writes illegal values to a sensitive location, then restores the original values before terminating. Also, we could not say anything about non-terminating terms: the applet could perform illegal writes, then enter an infinite loop to avoid detection. By providing the store control φ in advance, we ensure that the first write error will be detected immediately and reported as the `err` result. Unfortunately, this provides no way to control stores to locations created during the evaluation (rule 8 chooses these locations outside of $\mathrm{Dom}(\varphi)$, meaning that writes to these locations will be free): only preexisting locations can be sensitive. (This can be viewed as an inadequacy of big-step semantics, and a small-step, reduction-based semantics would fare better here. However, there are other features of big-step semantics that are crucial to our study: the clean separation between browser-supplied environment and applet-supplied source term, and the ability to interpret abstract type names by arbitrary sets of values.)

3 Reachability-based security

3.1 Simple reachability

The first security property for our calculus formalizes the idea that an applet can only write locations that are reachable in the initial environment in which it executes, or that are created during the applet's execution. For instance, if the references representing files are not reachable from the execution environment given to applets, then no applet can write to a file.

Reachability, here, is to be understood in the garbage collection sense: a location is reachable if there exists a path in the memory graph from the initial environment to the location, following one or several pointers. More formally, we define the set $RL(v, s)$ of locations reachable from a value v in a store s by the following equations:

$$\begin{aligned} RL(b, s) &= \emptyset \\ RL(\lambda x.a[e], s) &= RL(e, s) \\ RL((v_1, v_2), s) &= RL(v_1, s) \cup RL(v_2, s) \\ RL(\ell, s) &= \{\ell\} \cup RL(s(\ell), s) \end{aligned}$$

Normal rules:

$$\varphi, e, s \vdash x \rightarrow e(x)/s \quad (1) \qquad \varphi, e, s \vdash b \rightarrow b/s \quad (2) \qquad \varphi, e, s \vdash \lambda x.a \rightarrow \lambda x.a[e]/s \quad (3)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow \lambda x.a'[e']/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2 \qquad \varphi, e'\{x \leftarrow v_2\}, s_2 \vdash a' \rightarrow r}{\varphi, e, s \vdash a_1(a_2) \rightarrow r} \quad (4)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2}{\varphi, e, s \vdash (a_1, a_2) \rightarrow (v_1, v_2)/s_2} \quad (5)$$

$$\frac{\varphi, e, s \vdash a \rightarrow (v_1, v_2)/s'}{\varphi, e, s \vdash \mathtt{fst}(a) \rightarrow v_1/s'} \quad (6) \qquad \frac{\varphi, e, s \vdash a \rightarrow (v_1, v_2)/s'}{\varphi, e, s \vdash \mathtt{snd}(a) \rightarrow v_2/s'} \quad (7)$$

$$\frac{\varphi, e, s \vdash a \rightarrow v/s' \qquad \ell \notin \mathrm{Dom}(s') \cup \mathrm{Dom}(\varphi)}{\varphi, e, s \vdash \mathtt{ref}(a) \rightarrow \ell/s'\{\ell \leftarrow v\}} \quad (8) \qquad \frac{\varphi, e, s \vdash a \rightarrow \ell/s'}{\varphi, e, s \vdash\, !a \rightarrow s'(\ell)/s'} \quad (9)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow \ell/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2 \qquad \ell \notin \mathrm{Dom}(\varphi) \text{ or } v_2 \in \varphi(\ell)}{\varphi, e, s \vdash (a_1 := a_2) \rightarrow \mathtt{()}/s_2\{\ell \leftarrow v_2\}} \quad (10)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow \ell/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2 \qquad \ell \in \mathrm{Dom}(\varphi) \text{ and } v_2 \notin \varphi(\ell)}{\varphi, e, s \vdash (a_1 := a_2) \rightarrow \mathtt{err}} \quad (11)$$

Error propagation rules:

$$\frac{\varphi, e, s \vdash a_1 \rightarrow \mathtt{err}}{\varphi, e, s \vdash a_1(a_2) \rightarrow \mathtt{err}} \quad (12) \qquad \frac{\varphi, e, s \vdash a_1 \rightarrow v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow \mathtt{err}}{\varphi, e, s \vdash a_1(a_2) \rightarrow \mathtt{err}} \quad (13) \qquad \frac{\varphi, e, s \vdash a_1 \rightarrow \mathtt{err}}{\varphi, e, s \vdash (a_1, a_2) \rightarrow \mathtt{err}} \quad (14)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow \mathtt{err}}{\varphi, e, s \vdash (a_1, a_2) \rightarrow \mathtt{err}} \quad (15) \qquad \frac{\varphi, e, s \vdash a \rightarrow \mathtt{err}}{\varphi, e, s \vdash \mathtt{fst}(a) \rightarrow \mathtt{err}} \quad (16) \qquad \frac{\varphi, e, s \vdash a \rightarrow \mathtt{err}}{\varphi, e, s \vdash \mathtt{snd}(a) \rightarrow \mathtt{err}} \quad (17)$$

$$\frac{\varphi, e, s \vdash a \rightarrow \mathtt{err}}{\varphi, e, s \vdash \mathtt{ref}(a) \rightarrow \mathtt{err}} \quad (18) \qquad \frac{\varphi, e, s \vdash a \rightarrow \mathtt{err}}{\varphi, e, s \vdash\, !a \rightarrow \mathtt{err}} \quad (19) \qquad \frac{\varphi, e, s \vdash a_1 \rightarrow \mathtt{err}}{\varphi, e, s \vdash (a_1 := a_2) \rightarrow \mathtt{err}} \quad (20)$$

$$\frac{\varphi, e, s \vdash a_1 \rightarrow v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \rightarrow \mathtt{err}}{\varphi, e, s \vdash (a_1 := a_2) \rightarrow \mathtt{err}} \quad (21)$$

Figure 1: Evaluation rules

$$RL(e, s) \quad = \quad \bigcup_{x \in \mathrm{Dom}(e)} RL(e(x), s)$$

The definition above is not well-founded by induction on v, since in the fourth case the value $s(\ell)$ is arbitrarily large and may contain ℓ again. It should be viewed as a fixpoint equation $RL = F(RL)$, where F is an increasing operator; RL is, then, the smallest fixpoint of that operator.

If p is a set of sensitive locations, we define $Prot(p)$ as the store control that maps locations $\ell \in p$ to $\emptyset$, thus disallowing all writes on ℓ, and is undefined on locations $\ell \notin p$. We can now formulate the first security property:

Security property 1 *Let p be a set of sensitive locations. If $p \cap RL(e, s) = \emptyset$, then for all applets a, we have $Prot(p), e, s \vdash a \not\rightarrow \mathtt{err}$.*

In other words, if none of the locations in p is reachable from the initial environment e and store s, then no applet a can trigger an error by writing to a location in p: the applet will either terminate normally or loop. The proof of this property is a simple inductive argument on the evaluation derivation, using the following proposition as the induction hypothesis:

Proposition 1 *If $p \cap RL(e, s) = \emptyset$ and $Prot(p), e, s \vdash a \rightarrow r$, then $r \neq \mathtt{err}$. Instead, $r = v/s'$ for some v and s'. Moreover, $p \cap RL(v, s') = \emptyset$, and for all values w such that $p \cap RL(w, s) = \emptyset$, we have $p \cap RL(w, s') = \emptyset$.*

It is important to remark that even if property 1 makes no hypothesis about applet a being well-typed, it is crucial that a be well-typed in some type system, in order to guarantee that its evaluation on an actual processor adheres to the rules given in figure 1. In particular, the actual evaluation of a must never use the bit pattern representing an integer as if it were a store location; otherwise, the applet could write to arbitrary locations. We rely on a being well-typed in any sound type system (such as the one presented below in section 4) to ensure that this cannot happen.

Property 1, though simple and already well-known in the field of garbage collection [19], establishes a number of properties without which no security is possible at all. First, our language has safe pointers: locations cannot be forged by casting well-chosen integers. Second, automatic memory management (garbage collection) is feasible and does not weaken security: a location that becomes unreachable remains unreachable; moreover, newly allocated locations are always initialized; therefore, unreachable locations can be reused safely. Third, the language enforces lexical scoping: the execution of the applet depends only on the environment in which it proceeds; the applet does not have access to the full execution environment of the browser — as would be the case in a dynamically-scoped language, such as Emacs Lisp, or a language with special constructs to access the environment of the caller, such as Tcl.

3.2 Reachability and procedural abstraction

In defining reachable locations, we have treated closures like tuples (as garbage collectors do): the locations reachable from $\lambda x.a[e]$ are those reachable from $e(y)$ for some y. There is, however, a big difference between closures and tuples. Tuples are passive data structures: any piece of code that has access to the tuple can then obtain pointers to the components of the tuple. Closures are active data structures: only the code part of the closure can access directly the data part of the closure (the values of the free variables); other code pieces can only apply the closure, but not get at the data part directly. In other words, the code part of a closure mediates access to the data part. This property is often referred to as procedural abstraction [27].

For instance, consider the following function, similar to many Unix system calls, where `uid` is a reference holding the identity of the caller (applet or browser):

```
λx. if !uid = browser
     then do something with high privileges
     else do something with low privileges
```

Assume this function is part of the applet environment, but not the reference `uid` itself. Then, there is no way that the applet can modify the location of `uid`, even though that location is reachable from the environment.

A less obvious example, where the reference `uid` is not trivially read-only, is the following function in the style of the Unix `setuid` system call:

```
λnewid. if !uid = browser
        then uid := newid
        else raise an error
```

Assuming `uid` is not initially `browser`, an applet cannot change `uid` by calling this function.

Procedural abstraction can be viewed as the foundation for access control lists and similar programming techniques where resources are systematically encapsulated inside functions that check the identity and credentials of the caller before granting access to the requested resources. For instance, a file opening function contains the whole data structure representing the file system in its closure, but grants access only to files with suitable permissions. Thus, while all files are reachable from the closure of the `open` function, only those that have suitable permissions can be modified by the caller.

To formalize these ideas, we set out to define the set of locations $ML(v, s)$ that are actually modifiable (not merely reachable) from a value v and a store s, and show that if a location ℓ is not in $ML(e, s)$, then any applet evaluated in the environment e does not write to ℓ. This result is stronger than property 1 because the location ℓ that is not modifiable from e in s can still be reachable (via closures) from e in s.

For passive data structures (locations, tuples), modifiability coincides with reachability: a location is modifiable from v/s if a sequence of `fst`, `snd`, and ! operations applied to v in s evaluates to that location. The difficult case is defining modifiable locations for a function closure. The idea is to consider all possible applications of the closure to an argument: a location ℓ is considered modifiable from the closure only if one of those applications writes to the location, or causes the location to become modifiable otherwise.

More precisely, let e_{api} be the execution environment given to applets, and let $c = \lambda x.a[e]$ be one of the closures contained in e_{api}. A location ℓ is modifiable from c in store s if there exists a value v such that the following conditions hold:

Condition 1. The application of the closure to v causes ℓ to be modified, i.e. $\{\ell \mapsto \emptyset\}, e\{x \leftarrow v\}, s \vdash a \rightarrow \texttt{err}$.

Example: Let e be the environment $[r \leftarrow \ell]$. Then, ℓ is reachable from $(\lambda x.\ r := x + 1)[e]$ in any store, since any application of the closure causes ℓ to be assigned.

Condition 2. Alternatively, the application of c to v does not modify ℓ, but returns a result value and a new store from which ℓ is modifiable.

Example: With e as in the previous example, ℓ is reachable from $(\lambda x.\ (r, 1))[e]$, since any application of that closure returns a pair with ℓ as first component. An applet can thus write to ℓ by applying the closure, extracting ℓ from the returned result, and assigning ℓ directly.

Condition 3. Alternatively, the application of c to v modifies a reference accessible to the applet in such a way that ℓ becomes modifiable from that reference.

Example: Consider $c = (\lambda p.\ p := r)[e]$, where $e = [r \leftarrow \ell]$ as usual. Applying c to a location ℓ', we obtain a modified store s' such that $s'(\ell') = \ell$, i.e. ℓ is modifiable from ℓ' in s' and can be modified by the applet.

Condition 4. Alternatively, the application of c to v assigns references internal to the browser functions in such a way that ℓ becomes modifiable from the environment e_{api} in the store s' at the end of the application.

Example: Consider the following environment e_{api}:

$$\begin{aligned} e_{api}(f) &= (\lambda x.\ n := !n + 1)[n \leftarrow \ell_n] \\ e_{api}(g) &= (\lambda x.\ \texttt{if}\ !n \geq 1\ \texttt{then}\ r := 0)[n \leftarrow \ell_n; r \leftarrow \ell] \end{aligned}$$

In a store s such that $s(\ell_n) = 0$, the location ℓ is not modifiable from $e_{api}(g)$. However, ℓ is modifiable from $e_{api}(f)$ in s, since one application of that closure returns a store s' such that $s'(\ell_n) = 1$, and in that store s', ℓ is modifiable from $e_{api}(g)$: any application of $e_{api}(g)$ with initial store s' writes to ℓ.

Condition 5. In conditions 1–4 above, it must be the case that the location ℓ found to be modifiable from the closure c is not actually modifiable from the argument v passed to the closure. Otherwise, we would not know whether the location really "comes from" the closure c, or is merely modified by the applet-provided argument v.

Example: Consider the higher-order function $c = (\lambda f.\ f(0))[\emptyset]$. If we apply c to $(\lambda n.\ r := n)[r \leftarrow \ell]$, we observe a write to location ℓ. However, ℓ should not be considered as modifiable from c, since it is also modifiable from the argument given to c.

As should now be apparent from the conditions 1–5 above, the notion of modifiability raises serious problems, both practical and technical. On the practical side, the set of modifiable locations $ML(v, s)$ is not computable from

v and s (unlike $RL(v,s)$): in the closure case, we must consider infinitely many possible arguments. Thus, we need general proof to determine $ML(v,s)$.

Moreover, modifiable locations cannot be determined locally. As condition 4 shows, the modifiable locations of a closure depend on the modifiable locations of all functions from the applet environment e_{api}. Thus, if we manage to determine $ML(e_{api},s)$, then add one single function to the applet environment, we must not only determine the modifiable locations from the new function, but also reconsider all other functions in the environment to see whether their modifiable locations have changed. This is clearly impractical. Hence, the notion of modifiability is not effective and is interesting only from a semantic viewpoint and as a guide to derive decidable security criteria in the sequel.

On the technical side, the conditions 1–5 above do not lead to a well-founded definition of the sets of modifiable locations $ML(v,s)$. The problem is condition 5 (the requirement that the location must not be modifiable from the argument given to the closure): viewing conditions 1–4 as a fixpoint equation for some operator, that operator is not increasing because of the negation in condition 5.

In appendix B, we tackle this problem and show that non-modifiable locations are indeed never modified in the particular case where the applet's environment e_{api} is well-typed and its type E_{api} does not contain any `ref` types, so that no references are exchanged directly between the applet and its environment. In the remainder of this paper, we abandon the notion of modifiability in its full generality, and develop more effective techniques to restrict writes to reachable locations, relying on type-based instrumentation of the browser code.

4 The type system

We now equip our language with a simple type system. The type system is based on simply-typed λ-calculus, with the addition of named, user-defined types. Despite its simplicity, this type system does not restrict drastically the expressiveness of our language. In particular, recursive functions can still be defined using references [30]. The type algebra is:

Types:	$\tau ::= \iota$	base type (`int`, `string`, etc.)
	$\mid t$	named type
	$\mid \tau_1 \rightarrow \tau_2$	function type
	$\mid \tau_1 \times \tau_2$	product type
	$\mid \tau$ `ref`	reference type

Conversely, we enrich the syntax of terms with two new constructs: explicit coercions to and from named types, and run-time validation of values of named types.

Terms:	$a ::= \ldots$	(as before)
	$\mid \tau(a)$	coercion to type τ
	$\mid OK_t(a)$	run-time validation at type t

The typing rules for the calculus are shown in figure 3, and the extra evaluation rules for the new constructs in figure 2.

4.1 Named types

The overall approach followed in section 5 is to identify groups of references having the same type, and apply them a given security policy. However, types from the simply-typed λ-calculus are too coarse for this purpose: references of type `string ref` can hold many different kinds of data, such as messages, filenames, and cryptographic keys; clearly, different security restrictions must be applied to these different kinds of strings.

To this end, we introduce named types t, defined by a mapping TD for type names to type expressions, stating that the type t is interconvertible with its implementation type $TD(t)$. For instance, we could introduce a type `filename`, defined to be equal to `string`. An expression e of type `string` cannot be used implicitly with type `filename`; an explicit injection into `filename`, written `filename`(e), is required. Conversely, accessing the string underlying an expression e' of type `filename` is achieved by `string`(e'). (See rules 35 and 36.) In this respect, our named types behave very much like the `is new` type definition in Ada, and unlike type abbreviations in ML. Making the coercions explicit facilitates the definition of the program transformations in section 5, ensuring in particular that each term has a unique type.

The mapping TD of type definitions is essentially global: type definitions local to an expression are not supported. Still, it is possible to type-check some terms against a set of type definitions TD' that is a strict subset of TD, thus rendering the named types not defined in TD' abstract in that term. We will use this facility in section 5.2 and 5.3 to make named types abstract in the applet.

4.2 Run-time validation of values

The other unusual feature of our type system is the family of operators OK_t (one for each named type t) used to perform run-time validation of their argument. For each named type t, we assume given a set $PV(t)$ of permitted values for type t. (We actually allow $PV(t)$ to be undefined for some types t, which we take to mean that all values of type t are valid.) The expression $OK_t(e)$ checks whether the value of e is in $PV(t)$; if yes, it returns the value unchanged; if no, it aborts the execution of the applet and reports an error. In the evaluation rules, the “yes” case corresponds to rule 23; there is no rule for the “no” case, meaning that no evaluation derivation exists if an OK_t test fails. In effect, we do not distinguish between failure of OK_t and non-termination. At any rate, we must not return the `err` result when OK_t fails: no write violation occurred.

By varying $PV(t)$, we can control precisely the values of type t that will pass run-time validation. For instance, $PV($`filename`$)$ could consist of all strings referencing files under the applet's temporary directory `/tmp/applet.`x, a new directory that is created empty at the beginning of the applet's execution. Combined with the techniques described in section 5, this would ensure that only files in this temporary directory can be accessed by the applet. Similarly, the set $PV($`widget`$)$ could consist of all widget descriptors referring to widgets that are children of the applet's top widget, thus preventing the applet from interacting with widgets belonging to the browser. Other examples of run-time validation include checking cryptographic signatures on data structures that the applet must carry around without tampering with.

In practice, validation $OK_t(e)$ involves not only the value of its argument e, but also external information such as the identity of the principal, and possibly user replies to dialog boxes. A typical example is the Java `SecurityManager` class, which determines the identity of the principal by inspection of the call stack. For simplicity, we still write OK_t as a function of the value of its argument.

The evaluation rule for OK_t assumes of course that mem-

$$\frac{\varphi, e, s \vdash a \to r}{\varphi, e, s \vdash \tau(a) \to r}\ (22) \qquad \frac{\varphi, e, s \vdash a \to v/s' \quad t \in \mathrm{Dom}(PV) \text{ implies } v \in PV(t)}{\varphi, e, s \vdash OK_t(a) \to v/s'}\ (23) \qquad \frac{\varphi, e, s \vdash a \to \mathtt{err}}{\varphi, e, s \vdash OK_t(a) \to \mathtt{err}}\ (24)$$

Figure 2: Extra evaluation rules for coercions and run-time checks

$$E \vdash x : E(x)\ (25) \qquad E \vdash b : \mathit{Typeof}(b)\ (26)$$

$$\frac{E\{x \leftarrow \tau'\} \vdash a : \tau}{E \vdash \lambda x.a : \tau' \to \tau}\ (27) \qquad \frac{E \vdash a_1 : \tau' \to \tau \quad E \vdash a_2 : \tau'}{E \vdash a_1(a_2) : \tau}\ (28)$$

$$\frac{E \vdash a_1 : \tau_1 \quad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ (29) \qquad \frac{E \vdash a : \tau_1 \times \tau_2}{E \vdash \mathtt{fst}(a) : \tau_1}\ (30) \qquad \frac{E \vdash a : \tau_1 \times \tau_2}{E \vdash \mathtt{snd}(a) : \tau_2}\ (31)$$

$$\frac{E \vdash a : \tau}{E \vdash \mathtt{ref}(a) : \tau\ \mathtt{ref}}\ (32) \qquad \frac{E \vdash a : \tau\ \mathtt{ref}}{E \vdash !a : \tau}\ (33) \qquad \frac{E \vdash a_1 : \tau\ \mathtt{ref} \quad E \vdash a_2 : \tau}{E \vdash (a_1 := a_2) : \mathtt{unit}}\ (34)$$

$$\frac{E \vdash a : \tau \quad \tau = TD(t)}{E \vdash t(a) : t}\ (35) \qquad \frac{E \vdash a : t \quad \tau = TD(t)}{E \vdash \tau(a) : \tau}\ (36) \qquad \frac{E \vdash a : t}{E \vdash OK_t(a) : t}\ (37)$$

Figure 3: Typing rules

bership in $PV(t)$ is decidable. This raises obvious difficulties if t stands for a function type, at least if the domain type is infinite. Difficulties for defining $PV(t)$ also arise if t is a reference type: checking the current contents of the references offers no guarantees with respect to future modifications; checking the locations of the references against a fixed set of locations is very restrictive. For those reasons, we restrict ourselves to types t that are defined as algebraic datatypes: type expressions obtained by combining base types with datatype constructors such as `list` or tuples, but not with `ref` nor the function arrow.

4.3 Type soundness

To relate the typing rules to the dynamic semantics, we define a semantic typing relation $S \models v : \tau$ saying whether the value v is semantically a correct value for type τ. The S component is a store typing, associating types to store locations. We simultaneously extend the $\models$ relation to stores and store typings ($\models s : S$), and to evaluation environments and typing environments ($S \models e : E$). The definition of $\models$ is shown below, and is completely standard [30, 17].

- $S \models b : \iota$ if $\mathit{Typeof}(b) = \iota$
- $S \models v : t$ if $S \models v : TD(t)$
- $S \models \lambda x.a[e] : \tau_1 \to \tau_2$ if there exists a typing environment E such that $S \models e : E$ and $E \vdash \lambda x.a : \tau_1 \to \tau_2$
- $S \models (v_1, v_2) : \tau_1 \times \tau_2$ if $S \models v_1 : \tau_1$ and $S \models v_2 : \tau_2$
- $S \models \ell : \tau\ \mathtt{ref}$ if $\tau = S(\ell)$
- $S \models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$ and for all $x \in \mathrm{Dom}(e)$, $S \models e(x) : E(x)$
- $\models s : S$ if $\mathrm{Dom}(s) = \mathrm{Dom}(S)$ and for all $\ell \in \mathrm{Dom}(s)$, $S \models s(\ell) : S(\ell)$.

Using the semantic typing relations defined above, we then have the familiar strong soundness property below for the type system. We say that a store typing S' extends S if $\mathrm{Dom}(S') \supseteq \mathrm{Dom}(S)$, and for all $\ell \in \mathrm{Dom}(S)$, we have $S'(\ell) = S(\ell)$. Remark that semantic typing is stable under store extension: if $S \models v : \tau$ and S' extends S, we also have $S' \models v : \tau$.

Proposition 2 (Type soundness) *Assume $E \vdash a : \tau$ and $\models s : S$ and $S \models e : E$. If $\varphi, e, s \vdash a \to v/s'$, then there exists a store typing S' extending S such that $S' \models v : \tau$ and $\models s' : S'$.*

Proof: The proof is a simple inductive argument on the evaluation derivation; see [17, prop. 3.6] for details. □

5 Type-based security properties

The type-based security properties developed in this section are based on a common idea: assuming all sensitive references have types of the form t `ref`, we instrument the functions composing the execution environment by inserting OK_t run-time checks at certain program points, in order to prevent illegal writes to references of type t `ref`. The applets themselves are not instrumented, of course, since their source code is generally unavailable. All we know about the applet is that it is well typed in a given typing environment and set of type definitions. It is the combination of this well-typing with the instrumented environment functions that guarantees security.

5.1 Instrumentation of writes and procedural abstraction

The first transformation we consider inserts an OK_t check before any write to a reference of type t `ref` with $t \in \mathrm{Dom}(PV)$. This way, we are certain that if a sensitive reference has type t `ref`, the environment functions will always store in it values that belong to $PV(t)$. Formally, we define the instrumentation scheme IW, operating on terms a^τ annotated with their type τ, as follows:

$$IW((a^{t\ \mathtt{ref}} := b^t)^{\mathtt{unit}}) \quad = \quad IW(a^{t\ \mathtt{ref}}) := OK_t(IW(b^t)) \quad \text{if } t \in \mathrm{Dom}(PV)$$

On other kinds of terms, IW is a simple morphism, e.g. $IW((\lambda x.a^\tau)^{\sigma\to\tau}) = \lambda x.IW(a^\tau)$, etc.

It is easy to show that write errors cannot happen inside instrumented terms. Given the permitted values PV and a store typing S, we define $Prot(PV,S)$ as the store control that restricts references of type t `ref`, $t \in \mathrm{Dom}(PV)$ to values in $PV(t)$, and allows arbitrary writes to other references: $Prot(PV,S)(\ell) = PV(t)$ if $S(\ell) = t$ and $t \in \mathrm{Dom}(PV)$, and $Prot(PV,S)(\ell)$ is undefined otherwise.

Proposition 3 *Assume* $E \vdash (a^{\tau\ \mathtt{ref}} := b^\tau) : \mathtt{unit}$ *and* $S \models e : E$ *and* $\models s : S$. *Write* $\varphi = Prot(PV,S)$. *If* $\varphi, e, s \vdash IW(a^{\tau\ \mathtt{ref}}) \not\to \mathtt{err}$ *and* $\varphi, e, s \vdash IW(b^\tau) \not\to \mathtt{err}$, *then* $\varphi, e, s \vdash IW(a^{\tau\ \mathtt{ref}} := b^\tau) \not\to \mathtt{err}$.

Proof: If $\tau = t$ for some named type $t \in \mathrm{Dom}(PV)$, then by definition of the instrumentation scheme, the right-hand side of the assignment is of the form $OK_t(b')$ for some b', which can only evaluate to an element of $PV(t)$ by rule 23. Hence, the assignment is valid with respect to $Prot(PV,S)$. If τ is not a t type or is outside $\mathrm{Dom}(PV)$, by proposition 2, $IW(a)$ evaluates to a location ℓ which does not have a type of the form t `ref`, $t \in \mathrm{Dom}(PV)$, and therefore such that $Prot(PV,S)(\ell)$ is undefined; hence, no write error occurs either. □

Proposition 3 only provides half of the security property: it shows that writes in instrumented code are safe, but only the execution environment contains instrumented code; the applet code is not instrumented and could therefore perform illegal writes to sensitive locations, if it could access those locations. In other terms, we must make sure that all sensitive locations are encapsulated inside functions, as in section 3.2. To this end, we will restrict the type E_{api} of the applet's execution environment e_{api} to ensure that sensitive references cannot "leak" into the applet, and be assigned illegal values there. There are several ways by which a sensitive reference of type t `ref` could leak into an applet:

- The reference is exported directly in the environment, e.g. $E_{api}(x) = t$ `ref` or $E_{api}(x) = \mathtt{int} \times (t\ \mathtt{ref})$.
- The reference is returned by one of the functions of the environment, e.g. $E_{api}(f) = \mathtt{int} \to t\ \mathtt{ref}$.
- The environment contains a higher-order function such as $E_{api}(h) = (t\ \mathtt{ref} \to \mathtt{int}) \to \mathtt{int}$. The applet could get access to a sensitive reference if h passes one to its functional argument, which can be provided by the applet.
- The environment contains a function taking as argument an applet-provided reference to a t `ref`, e.g. $E_{api}(f) = t\ \mathtt{ref}\ \mathtt{ref} \to \mathtt{unit}$. The environment function f could then store a sensitive location into that t `ref ref`, from which the applet can recover the sensitive location later.

We rule out all these cases by simply requiring that no type t `ref` occurs (at arbitrary depths) in E_{api}. This leads to the following security property:

Security property 2 *Assume* $S \models e_{api} : E_{api}$ *and* $\models s : S$. *Further assume that* E_{api} *contains no occurrence of* t `ref` *for any* t, *and that all function closures in* e *and* s *have been instrumented with the* IW *scheme (that is,* e *and* s *are obtained by evaluating source terms instrumented with* IW*). Then, for every applet* a *well-typed in* E, *we have* $Prot(PV,S), s, e \vdash a \not\to \mathtt{err}$.

Proof: We consider all assignments to references that occur in the evaluation derivation for a. By proposition 3, assignments performed by environment functions cannot cause a write error, since the right-hand side has been instrumented by IW. For assignments performed by the applet, we use a containment lemma developed in appendix A to show that the reference being assigned cannot belong to $\mathrm{Dom}(Prot(PV,S))$. More precisely, we apply appendix A with T being the set of all type expressions where t `ref` occurs, L_{app} being the set of locations with types in T allocated by the applet, and L_{env} being the set of locations with types in T allocated by environment functions or initially present in e_{api}. The containment lemma (proposition 7) then shows that the location being assigned does not belong to L_{env}. Moreover, by construction of $Prot$ and L_{env}, we have $\mathrm{Dom}(Prot(PV,S)) \subseteq L_{env}$. Thus, assignments performed by the applet do not cause write violations either. Hence, the applet cannot evaluate to `err`. □

The requirement that no t `ref` occurs in E_{api} is clearly too strong: nothing wrong could happen if, for instance, one of the environment functions has type $t\ \mathtt{ref} \to \mathtt{unit}$ (the t `ref` argument is provided by the applet). We conjecture that it suffices to require that type t `ref` does not occur in E_{app} at a positive occurrence nor under a `ref` constructor. However, our proof of property 2, and in particular the crucial containment lemma 7, does not extend to this weaker hypothesis.

5.2 Instrumentation of coercions and type abstraction

Instead of putting run-time checks on writes to references of type t `ref`, we can ensure that all values of type $t \in \mathrm{Dom}(PV)$ that flow through the applet's execution environment always belong to $PV(t)$. This way, values stored in a t `ref` will automatically satisfy $PV(t)$ as well. This is achieved by adding checks to all creations of values of type $t \in \mathrm{Dom}(PV)$ in the execution environment, i.e. to coercions of the form $t(a)$, following the instrumentation scheme IC below:

$$IC(t(a)) \quad = \quad OK_t(t(IC(a))) \quad \text{if } t \in \mathrm{Dom}(PV)$$

Of course, this is not enough: the applet could forge unchecked values of type t, by direct coercion from t's implementation type, and pass them to environment functions. Hence, we also need to make the types $t \in \mathrm{Dom}(PV)$ abstract in the applet, by type-checking it with a set of type definitions TD' obtained from TD by removing the definitions of the types $t \in \mathrm{Dom}(PV)$. Then, for any $t \in \mathrm{Dom}(PV)$, the only values of type t that can be manipulated by the applet have been created and checked by the environment.

To capture the run-time behavior of instrumented terms, we introduce a variant of the semantic typing predicate, written $PV, S \models v : \tau$, which is similar to the predicate $S \models v : \tau$ introduced in section 4.3, with the difference that a value v belongs to a named type t only if $v \in PV(t)$ in addition to v belonging to the definition $TD(t)$ of t:

- $PV, S \models b : \iota$ if $\mathit{Typeof}(b) = \iota$
- $PV, S \models v : t$ if $PV, S \models v : TD(t)$ and $t \in \mathrm{Dom}(PV)$ implies $v \in PV(t)$
- $PV, S \models \lambda x.a[e] : \tau_1 \to \tau_2$ if there exists a typing environment E such that $PV, S \models e : E$ and $E \vdash \lambda x.a : \tau_1 \to \tau_2$
- $PV, S \models (v_1, v_2) : \tau_1 \times \tau_2$ if $PV, S \models v_1 : \tau_1$ and $PV, S \models v_2 : \tau_2$
- $PV, S \models \ell : \tau$ `ref` if $\tau = S(\ell)$
- $PV, S \models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$ and for all $x \in \mathrm{Dom}(e)$, $PV, S \models e(x) : E(x)$
- $PV \models s : S$ if $\mathrm{Dom}(s) = \mathrm{Dom}(S)$ and for all $\ell \in \mathrm{Dom}(s)$, $PV, S \models s(\ell) : S(\ell)$.

We then have the following characterization of the the behavior of terms instrumented with IC:

Proposition 4 *Assume $E \vdash a : \tau$ and $PV, S \models e : E$ and $PV \models s : S$. Further assume that all closures contained in e and s have function bodies instrumented with IC. If $\mathit{Prot}(PV, S), e, s \vdash IC(a) \to r$, then $r \neq$ `err`; instead, r is of the form v/s', and there exists a store typing S' extending S such that $PV, S' \models v : \tau$ and $PV \models s' : S'$.*

Compared with proposition 2, we now use the more restrictive interpretation of named types PV, and require that all terms occurring in the evaluation derivation have been instrumented with IC. The proof is essentially identical to that of proposition 2.

Notice that an applet a well-typed within the restricted set TD' of type definitions contains no coercions $t(a)$ for any $t \in \mathrm{Dom}(PV)$, hence is equal to $IC(a)$. Thus, proposition 4 applies not only to the terms from the execution environment, but also to the applet itself. In a sense, this proposition can be viewed as a parametricity result, in that it shows soundness for arbitrary interpretations PV of the types left abstract in TD'. From this remark, we immediately obtain the following security property:

Security property 3 *Let e be the execution environment for applets, and s the initial store. Assume that all function closures in e and s have been instrumented with the IC scheme (that is, e and s are obtained by evaluating source terms instrumented with IC). Assume $PV \models s : S$ and $PV, S \models e : E$. Then, for every applet a well-typed in E and in the restricted set TD' of type definitions, we have $\mathit{Prot}(PV, S), s, e \vdash a \not\to$ `err`.*

One practice that property 3 formally justifies is systems based on capabilities: by making the type of capabilities abstract to the applets, run-time security checks are necessary only at points where new capabilities are constructed and returned to the applet; capabilities presented by the applet can then be trusted without further checks.

Unlike property 2, property 3 does not require that types t `ref` do not occur in the typing environment E. Indeed, once t is made abstract, it is perfectly safe to make references of type t `ref` accessible to the applet: the applet can then write to them, but only write safe values of type t. Hence, sensitive references no longer need to be systematically wrapped inside functions. As Reynolds points out [27], type abstraction and procedural abstraction are two orthogonal ways to protect data.

Another advantage of the approach described in this section over the instrumentation of writes described in section 5.1 is that it often leads to fewer run-time checks. In particular, checks at coercions can sometimes be proven redundant and therefore can be eliminated. Consider the following function that adds a `.old` suffix to a file name:

λ`f.`$\mathit{OK}_{\texttt{filename}}$`(filename(concat(string(f),".old")))`

With the definition of $PV($`filename`$)$ given in section 4.2, it is easy to show that if f belongs to $PV($`filename`$)$, then so does the concatenation of f and `.old`. Hence, the OK test can be removed.

Of course, not all run-time tests can be removed this way: consider what happens if the suffix was given as argument:

λ`f.`λ`s.`$\mathit{OK}_{\texttt{filename}}$`(filename(concat(string(f),s)))`

and the applet passes a suffix `s` starting with “`../`”.

5.3 Instrumenting coercions without type abstraction

In some cases, the types $t \in \mathrm{Dom}(PV)$ cannot be made abstract in the applet, e.g. because it would make writing the applet too inconvenient, or entail too much run-time overhead. We can adapt the approach presented in section 5.2 to these cases, by reverting to procedural abstraction and putting checks not only at coercions, but also on all values of types $t \in \mathrm{Dom}(PV)$ that come from the applet. (This matches current practice in Unix kernels, where parameters to system calls are always checked for validity on entrance to the system call.)

This is achieved by a standard wrapping scheme applied to all functions of the execution environment, inserting OK_t coercions at all negative occurrences of types $t \in \mathrm{Dom}(PV)$. For instance, if the execution environment needs to export a function $f : t \to t$, it will actually export the function $\lambda x.f(OK_t(x))$, which validates its argument before passing it to the original function.

We formalize these ideas in a slightly different way, in order to build upon the results of section 5.2. Start from an applet environment defined by top-level bindings of the form

$$\texttt{let}\ f_i : \tau_i = a_i$$

We assume given a set TD' of type definitions against which the a_i and the applets are type-checked, and a valuation PV' assigning permitted values to named types in TD'.

We first associate a new named type $\hat{t}$ to each sensitive type $t \in \mathrm{Dom}(PV')$. The type $\hat{t}$ is defined as synonymous with t, and is intended to represent those values of type t that have passed run-time validation. We define the $\hat{t}$ types by taking

$$TD = TD' \oplus [\hat{t} \mapsto t \mid t \in \mathrm{Dom}(PV')]$$

and restrict the values they can take using the valuation PV defined by $PV(\hat{t}) = PV'(t)$ and PV undefined on other types.

Let Σ be the substitution $\{t \leftarrow \hat{t} \mid t \in \mathrm{Dom}(PV')\}$. We transform the bindings for the applet environment as follows:

$$\texttt{let}\ f_i : \tau_i = W^+(IC(\Sigma(a_i)) : \tau_i)$$

That is, we rewrite the terms a_i to use the type $\hat{t}$ instead of t for all $t \in \mathrm{Dom}(PV')$; then apply the IC instrumentation scheme to it, thus adding an $OK_{\hat{t}}$ check to each coercion $\hat{t}(a)$; finally, apply the W^+ wrapping scheme to the instrumented term, in order to perform both validation and coercion from t to $\hat{t}$ on entrance, and the reverse coercion from $\hat{t}$ to t on exit. Wrapping is directed by the expected type for its result, and is contravariant with respect to function types. We thus define both a wrapping scheme W^+ for positive occurrences of types and another W^- for negative occurrences.

$$\begin{array}{rcl}
W^+(a:\iota) & = & W^-(a:\iota) \quad = \quad a \\
W^+(a:t) & = & t(a) \text{ if } t \in \mathrm{Dom}(PV) \\
W^-(a:t) & = & OK_{\hat{t}}(\hat{t}(a)) \quad \text{if } t \in \mathrm{Dom}(PV) \\
W^+(a:t) & = & W^-(a:t) \quad = \quad a \quad \text{if } t \notin \mathrm{Dom}(PV) \\
W^+(a:\tau_1 \times \tau_2) & = & (W^+(\mathtt{fst}(a):\tau_1), W^+(\mathtt{snd}(a):\tau_2) \\
W^-(a:\tau_1 \times \tau_2) & = & (W^-(\mathtt{fst}(a):\tau_1), W^-(\mathtt{snd}(a):\tau_2) \\
W^+(a:\tau_1 \rightarrow \tau_2) & = & \lambda x.W^+(a(W^-(x:\tau_1)):\tau_2) \\
W^-(a:\tau_1 \rightarrow \tau_2) & = & \lambda x.W^-(a(W^+(x:\tau_1)):\tau_2) \\
W^+(a:\tau\ \mathtt{ref}) & = & W^-(a:\tau\ \mathtt{ref}) \quad = \quad a \\
 & & \text{if no } t \in \mathrm{Dom}(PV) \text{ occurs in } \tau
\end{array}$$

Wrapping is not defined on reference types containing a $t \in \mathrm{Dom}(PV)$ because there is no way to validate these references so that they are protected against future modifications.

It is easy to see that the transformed bindings are well-typed: $\Sigma(a_i)$ has type $\Sigma(\tau_i)$; the IC instrumentation preserves typing; the W^+ wrapping applied to a term of type $\Sigma(\tau_i)$ returns a term of type τ_i, as shown by a simple induction over τ_i.

Moreover, the right-hand sides of the bindings always perform an $OK_{\hat{t}}$ check before each coercion to a type $\hat{t}$: this is ensured by the IC instrumentation for the coercions initially in $\Sigma(a_i)$, and by definition of the wrapping scheme for the coercions introduced by the wrapping.

Finally, the applets themselves are still type-checked in the original set TD' of type definitions, in which the types $\hat{t}$ are not defined, and thus abstract for the applet.

We are therefore back to the situation studied in section 5.2: the types $\hat{t}$ are abstract in the applets and all coercions to $\hat{t}$ are instrumented in the applet environment. Thus, by property 3, we obtain that the values of references with types $\hat{t}$ `ref` always remain within $PV(\hat{t}) = PV'(t)$ during the execution of any well-typed applet.

Security property 4 *Let e be the execution environment for applets and s the initial store. Assume that e and s are obtained by evaluating a set of transformed bindings* `let` $f_i : \tau_i = W^+(IC(\Sigma(a_i)) : \tau_i)$ *as described above. Assume* $PV \models s : S$ *and* $PV, S \models e : E$*. Then, for every applet a well-typed in E and in the initial set* TD' *of type definitions, we have* $Prot(PV, S), s, e \vdash a \not\rightarrow \mathtt{err}$.

6 Connections with object-oriented languages

Although the language used for this work is functional, the techniques developed here translate reasonably well to object-oriented languages. Procedural abstraction as presented in section 3.2 corresponds most closely to Smalltalk-style private instance variables: just as variables in a closure environment can only be accessed by the code associated with that closure, private instance variables of a Smalltalk object can only be accessed by the methods of that object. Java does not offer a strictly equivalent mechanism: the `private` modifier makes instance variables accessible not only to the methods of the object, but also to methods of other objects of the same class. Still, the visibility rules associated with the package mechanism and the `private` and `protected` modifiers also ensure some degree of procedural abstraction, since they restrict the set of methods that can access a given instance variable.

Similarly, type abstraction as exploited in section 5.3 corresponds most closely to `final` classes in Java. A `final` class containing only `private` fields and having a `private` constructor offers the same level of guarantees as our abstract types: the applet cannot create directly an object of that class, nor tamper with the fields of an existing object. In non-`final` classes, the visibility rules might still ensure some degree of type abstraction, though it is unclear how much is guaranteed. Subtype polymorphism in systems such as $F_{<:}$ [7] also provide some amount of type abstraction, but this is not so in Java because the actual type of an object can be tested at run-time (downcasts).

7 Related work

7.1 Type systems for security

The work most closely related to ours is the recent formulations of Denning's information flow approach to security [10, 9] as non-standard type systems by Palsberg and Ørbaek [24], Volpano and Smith [31, 32], and Heintze and Riecke [14]. The main points of comparison with our work are listed below.

Information flow vs. integrity: The type systems developed in previous works all focus on flow of information between high-security and low-security parts of a program. In particular, they allow high-security data to be exposed as long as no low-security code uses this data. In the context of applets, we have no control over the low-security code and are therefore forced to ensure that high-security data is never exposed directly. Thus, our work does not concentrate on information flow proper, but mainly on protecting sensitive data via procedural or type abstraction.

Imperative vs. purely functional programs: [24] and [14] consider purely functional languages in the style of the λ-calculus. This makes formulating the security properties delicate: [24] proves no security property properly speaking, only a subject reduction property that shows the internal consistency of the calculus, but not its relevance to security; [14] does show a non-interference property (that the value of a low-security expression is independent of the values of high-security parameters), but it is not obvious how this result applies to actual applet/browser interactions, especially input/output. Instead, we have followed [31, 32] and formulated our security policy in terms of in-place modifications on a store, which provides a fairly intuitive idea of what constitutes a security violation.

Run-time validation of data: Only [24] and our work consider the possibility of checking low-security data at run-time and promoting them to high security. In [31, 32, 14], once some data is labeled "low security", it remains so throughout the program and causes all data it comes in contact with to be marked "low security" as well. We believe that, in a typical applet/browser interaction, this policy leads to rejecting almost all applets as insecure. Run-time validation of untrusted data is essential in practice to allow a reasonable range of applets to run.

Subtyping vs. named types and coercions: All previous works consider type systems with subtyping, which provides a good match for the flow analysis approach they follow [23]. In contrast, we only use type synonyms with possibly checked coercions between a named type and its implementation type. However, the connections between subtyping and explicit coercions are well known [5], and we do not think this makes a major difference.

7.2 Security of applets

Concerning the security issues raised by applets in general, we are aware of case studies of security flaws [8], as well as informal descriptions of current and proposed security architectures for applets [12, 34, 28, 33]. Our work seems to be one of the first formal studies of applet security.

On the Java side, considerable effort has been expended in proving the soundness of the Java type system [11, 22] and of the JVM bytecode verifier [26]. Other aspects of Java that are equally important for security, such as formalizing the visibility rules and the encapsulation guarantees they provide, have only recently started to receive attention [16]. We are not aware of any formal description of security policies for Java applets.

Proof-carrying code [20, 21] provides an elegant framework to establish the safety of mobile code, but requires general proof from the applet's developer. Our approach lies at the other end of the complexity spectrum: all we require from the applet is that it is well typed in a simple, standard type system.

8 Concluding remarks

We have identified three basic techniques for enforcing a fairly realistic security policy for applets: lexical scoping, procedural abstraction, and type abstraction. These programming techniques are of course well known, but we believe that this work is the first to characterize precisely their implications for program security.

The techniques proposed here seem to match relatively well current practice in the area of Web applets. In particular, they account fairly well for Rouaix's implementation of safe libraries in the MMM browser [28]. Some aspects of MMM are still unaccounted for, such as the necessity of taking copies of mutable objects before validating them. Also, MMM applets have the ability to install functions in various stages of the browser (HTML display machine, document decoding), and this requires validation of these functions at times other than those suggested by our framework.

Our techniques put almost no constraints on the applets, except being well-typed in a simple, completely standard type system. The security effort is concentrated on the execution environment provided by the browser. Typing the applets in a richer type system, such as the type systems for information flow of [24, 31, 32, 14] or the effect and region system of [29], could provide more information on the behavior of the applet and enable more flexible security policies in the execution environment. However, it is probably impractical to rely on rich type systems for applets, because these type systems are not likely to be widely accepted by applet developers. Whether these rich type systems can be applied to the execution environment only is an interesting open question.

On the technical side, the proofs of the type-based security properties are variants of usual type soundness proofs. It would be interesting to investigate the security content of other classical semantic results such as representation independence and logical relations. Given the importance of communications between the applet and its environment, it could be worthwhile to reformulate our security results for a calculus of communicating processes [2, 1].

References

[1] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *CONCUR'97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, July 1997.

[2] J.-P. Banâtre and C. Bryce. A security proof system for networks of communicating processes. Research report 2042, INRIA, Sept. 1993.

[3] J.-P. Billon. Security breaches in the JDK 1.1 beta2 security API. Dyade, `http://www.dyade.fr/fr/actions/VIP/SecHole.html`, Jan. 1997.

[4] N. S. Borenstein. Email with a mind of its own: the Safe-Tcl language for enabled mail. In *IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications*, 1994.

[5] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[6] K. Brunnstein. Hostile ActiveX control demonstrated. *RISKS Forum*, 18(82), Feb. 1997.

[7] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[8] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy*. IEEE, 1996.

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

[10] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–242, 1976.

[11] S. Drossopoulou and S. Eisenbach. Java is type safe – probably. In *Proc. 11th European Conference on Object Oriented Programming*, June 1997.

[12] M. Erdos, B. Hartman, and M. Mueller. Security reference model for the Java Developer's Kit 1.0.2. JavaSoft, `http://java.sun.com/security/SRM.html`, Nov. 1996.

[13] J. Gosling and H. McGilton. The Java language environment – a white paper. JavaSoft, `http://java.sun.com/docs/white/langenv`, May 1996.

[14] N. Heinze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. Draft, available electronically, Mar. 1998.

[15] D. Hopwood. Java security bug (applets can load native methods). *RISKS Forum*, 17(83), Mar. 1996.

[16] T. Jensen, D. Le Métayer, and T. Thorn. A formalisation of visibility and dynamic loading in Java. Technical Report 1137, IRISA, Oct. 1997.

[17] X. Leroy. Polymorphic typing of an algorithmic language. Research report 1778, INRIA, 1992.

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[19] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Functional Programming Languages and Computer Architecture 1995*, pages 66–77. ACM Press, 1995.

[20] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. Symp. Operating Systems Design and Implementation*, 1996.

[21] G. C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[22] T. Nipkow and D. von Oheimb. JavaLight is type-safe — definitely. In *25th symposium Principles of Programming Languages* ACM Press, 1998.

[23] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *22nd symposium Principles of Programming Languages*, pages 367–378. ACM Press, 1995.

[24] J. Palsberg and P. Ørbaek. Trust in the λ-calculus. In *Symposium on Static Analysis '95*, volume 983 of *Lecture Notes in Computer Science*, pages 314–329. Springer-Verlag, 1995.

[25] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[26] Z. Qian. A formal specification of a large subset of Java Virtual Machine instructions. Draft, available electronically, Sept. 1997.

[27] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In C. Gunter and J. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 13–23. MIT Press, 1994.

[28] F. Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.

[29] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[30] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), 1990.

[31] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[32] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*, 1997.

[33] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. Technical report 546-97, Department of Computer Science, Princeton University, Apr. 1997.

[34] F. Yellin. Low level security in Java. In *Proceedings of the Fourth International World Wide Web Conference*. O'Reilly, 1995.

A Appendix: the containment lemma

In this appendix, we formalize the intuition that if the type of the applet environment does not contain certain `ref` types, then references of those types cannot be exchanged between the applet and the environment, and remain "contained" in one of them.

We annotate each source-language term a as coming either from the execution environment (a_{env}) or from the applet (a_{app}). We let m, n range over the two "worlds" *env* and *app*, and write $\overline{m}$ for the complement of m, i.e. $\overline{env} = app$ and $\overline{app} = env$.

Let T be a set of type expressions satisfying the following closure property: if $\tau \in T$, then all types τ' that contain τ as a sub-term also belong to T. (In section 5.1, we take T to be the set of all types containing an occurrence of t `ref`; in appendix B, T is the set of all types containing an occurrence of any `ref` type). We partition the set of locations into three countable sets:

- L_{app} is the set of locations with type τ `ref` $\in T$ that have been allocated by the applet;
- L_{env} is the set of locations with type τ `ref` $\in T$ that have been allocated by the environment (i.e. either initially present in the applet environment, or allocated by environment functions);
- L_{shared} is the set of locations whose type τ `ref` does not belong to T.

To ensure that locations allocated during evaluation are drawn from the correct set, we assume all source terms a^{τ}_{m} annotated with their static type τ and their world m and replace the evaluation rule for reference creation (rule 8) by the following rule:

$$\frac{\begin{array}{c}\varphi, e, s \vdash a^{\tau}_{m} \to v/s' \qquad \ell \notin \mathrm{Dom}(s') \cup \mathrm{Dom}(\varphi) \\ \ell \in L_{env} \text{ if } \tau\ \mathtt{ref} \in T \text{ and } m = env \\ \ell \in L_{app} \text{ if } \tau\ \mathtt{ref} \in T \text{ and } m = app \\ \ell \in L_{shared} \text{ if } \tau\ \mathtt{ref} \notin T\end{array}}{\varphi, e, s \vdash \mathtt{ref}(a)^{\tau\ \mathtt{ref}}_{m} \to \ell/s'\{\ell \leftarrow v\}} \quad (8')$$

It is easy to see that the modified rule produces the same evaluation derivations as the initial rule, up to a renaming of fresh locations: if $\varphi, e, s \vdash a \to r$ with the initial rule, then $\varphi, e, s \vdash a \to r'$ with the modified rule, where r' is identical to r up to a renaming of locations not in $\mathrm{Dom}(s) \cup \mathrm{Dom}(\varphi)$.

We say that a value v in a store s is contained in world m, and we write $C_m(v, s)$, if any source term operating on v in s can directly access only locations that are in L_m or L_{shared}, but not locations in $L_{\overline{m}}$. Formally, $C_m(v, s)$ is defined by case analysis on v, as follows:

- $C_m(b, s)$ is always true
- $C_m(\lambda x.a_n[e], s)$ if $C_n(e, s)$
- $C_m((v_1, v_2), s)$ if $C_m(v_1, s)$ and $C_m(v_2, s)$

- $C_m(\ell, s)$ if $l \notin L_{\overline{m}}$ and $C_m(s(\ell), s)$
- $C_m(e, s)$ if $C_m(e(x), s)$ for all $x \in \mathrm{Dom}(e)$.

Notice that for closures, it's the world n of the function body a_n that determines the containment of the closure environment, not the world m in which the closure value is being used. The reason is that the environment values can only be accessed by the function body, not arbitrary terms of world m. This is characteristic of procedural abstraction in the sense of section 3.2.

As usual, the definition of C above is not well-founded by induction on v. We view the equations above as fixpoint equations for an operator, which is increasing, and define C as the greatest fixpoint of that operator. (The smallest fixpoint is always false on value/store pairs that contain a cycle, which is not what we want; it's the greatest fixpoint that gives the expected behavior for C. See [30] for detailed explanations.)

Here are two important lemmas on the C predicate. First, assigning a value contained in world m to a location which is not in $L_{\overline{m}}$ preserves the containment of all other values.

Proposition 5 *Assume either $\ell \in L_m$ and $C_m(v, s)$, or $\ell \in L_{shared}$ and $C_{app}(v, s)$ and $C_{env}(v, s)$. Then, for all values w and worlds n, $C_n(w, s)$ implies $C_n(w, s\{\ell \leftarrow v\})$.*

Proof: The proof is a standard argument by coinduction, close to the proof of lemma 4.6 in [30]. The only non-trival case is $w = \ell$ (the modified location). By hypothesis $C_n(w, s)$, we have $\ell \notin L_{\overline{n}}$ and $C_n(s(\ell), s)$. Write $s' = s\{\ell \leftarrow v\}$. Notice that $s'(\ell) = v$. If $n = m$, we have $C_n(v, s)$ by assumption, hence $C_n(v, s')$ by the coinduction hypothesis, from which it follows that $C_n(\ell, s')$. If $n = \overline{m}$, since $\ell \notin L_{\overline{n}}$, we have $\ell \notin L_m$ and thus it must be the case that $\ell \in L_{shared}$ and $C_{app}(v, s)$ and $C_{env}(v, s)$ to comply with the assumptions of the proposition. Thus, we have $C_n(v, s)$ and we conclude as in the previous case. □

Second, values that belongs to a type $\tau \notin T$ can be exchanged between the *app* and *env* worlds without breaking containment.

Proposition 6 *Assume $S \models v : \tau$ and $\models s : S$. If $\tau \notin T$, then $C_m(v, s)$ implies $C_{\overline{m}}(v, s)$ for all worlds m.*

Proof: By structural induction on τ. If τ is a base type or a function type, then v is a base value or a closure, and the containment of v is independent of the world m. If τ is a product type $\tau_1 \times \tau_2$, the closure condition over T guarantees that $\tau_1 \notin T$ and $\tau_2 \notin T$; the result follows from the induction hypothesis. Finally, if τ is a `ref` type, we have $v = \ell$ and $\tau = S(\ell)\ \mathtt{ref}$. Since $\tau \notin T$, it follows that ℓ belongs to L_{shared}, but neither to L_{app} nor L_{env}. Hence, $l \notin L_m$. Moreover, $S(\ell) \notin T$ by the closure condition on T, hence $C_{\overline{m}}(s(\ell), s)$ by application of the induction hypothesis. It follows that $C_{\overline{m}}(\ell, s)$. □

We can now show that containment is preserved at each evaluation step:

Proposition 7 (Containment lemma) *Let E_{api} be the type of the execution environment for applets. Assume $E_{api}(x) \notin T$ for all $x \in \mathrm{Dom}(E_{api})$. Further assume $E \vdash a_m : \tau$ and $S \models e : E$ and $\models s : S$ and $C_m(e, s)$. If $\varphi, e, s \vdash a_m \to v/s'$, then $C_m(v, s')$, and for all values w and worlds n such that $C_n(w, s)$, we have $C_n(w, s')$.*

Proof: The proof is by induction on the evaluation derivation and case analysis on a. Notice that by proposition 2, we have the additional result that there exists a store typing S' extending S such that $S' \models v : \tau$ and $\models s' : S'$. This makes the semantic typing hypotheses go through the induction. The interesting cases are assignment and function application; the other cases are straightforward.

Assignment: a is $(a_1^{\sigma\ \mathtt{ref}} := a_2^{\sigma})$. We apply the induction hypothesis twice, obtaining

$$\begin{gathered}\varphi, e, s \vdash a_1 \to v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \to v_2/s_2 \\ C_m(v_1, s_2) \qquad C_m(v_2, s_2) \\ C_n(w, s) \text{ implies } C_n(w, s_2) \text{ for all } n, w \\ S_2 \models v_1 : \sigma\ \mathtt{ref} \qquad S_2 \models v_2 : \sigma \qquad \models s_2 : S_2.\end{gathered}$$

Hence, v_1 is a location ℓ, and since v_1 is contained in m, we have $\ell \notin L_{\overline{m}}$. Therefore, either $\ell \in L_m$ or $\ell \in L_{shared}$. But in the latter case, $\sigma \notin T$ by construction of L_{shared}, hence $C_m(v_2, s_2)$ implies $C_{\overline{m}}(v_2, s_2)$ as well by proposition 6. In both cases, the hypotheses of proposition 5 are met. Writing $s' = s_2\{\ell \leftarrow v_2\}$, we obtain the expected result: $C_n(w, s)$ implies $C_n(w, s')$ for all n, w. The other expected result, $C_m((), s')$, is trivial.

Application: a is $a_1^{\sigma \to \tau}(a_2^{\sigma})$. By applying the induction hypothesis twice, we obtain

$$\begin{gathered}\varphi, e, s \vdash a_1 \to v_1/s_1 \qquad \varphi, e, s_1 \vdash a_2 \to v_2/s_2 \\ C_m(v_1, s_2) \qquad C_m(v_2, s_2) \\ C_n(w, s) \text{ implies } C_n(w, s_2) \text{ for all } n, w \\ S_2 \models v_1 : \sigma \to \tau \qquad S_2 \models v_2 : \sigma \qquad \models s_2 : S_2.\end{gathered}$$

Hence, v_1 is a closure $\lambda x.a'_n[e']$, and the last evaluation rule used is rule 4.

If $n = m$ (intra-world call), we have $C_m(e'\{x \leftarrow v_2\}, s_2)$ as a consequence of $C_m(v_1, s_2)$ and $C_m(v_2, s_2)$, and the two conclusions follows easily from the induction hypothesis applied to the evaluation of a'_n.

If $n = \overline{m}$ (cross-world call), then the type $\sigma \to \tau$ of the function must occur as a sub-term of the typing environment E_{api}. Hence, $\sigma \notin T$ and $\tau \notin T$. Since the type of v_2 is not in T, by proposition 6 it follows that $C_n(v_2, s_2)$. Hence, $C_n(e'\{x \leftarrow v_2\}, s_2)$, and we can apply the induction hypothesis to the evaluation of a'_n: $\varphi, e'\{x \leftarrow v_2\}, s_2 \vdash a'_n \to v/s'$. The resulting value v is contained in world n and has type τ; applying again proposition 6, we get $C_m(v, s')$, which is the expected result. □

B Appendix: modifiable locations in the case of systematic procedural encapsulation

In this appendix, we formalize the notion of modifiable locations, as introduced in section 3.2, in the particular case where the applet environment e_{api} is well-typed and its type E_{api} does not contain any `ref` types. This is not to say that e_{api} is purely functional: the functions it provide may very well have side-effects and use references internally. Only, all those internal references must be encapsulated inside functions and never handed to the applet directly. This systematic procedural encapsulation does not reduce expressiveness in any significant way, and at any rate is a reasonable thing to do given that our goal is to characterize semantically procedural encapsulation.

$$
\begin{array}{rcl}
ML(b, s) & = & \emptyset \\
ML((v_1, v_2), s) & = & ML(v_1, s) \cup ML(v_2, s) \\
ML(\lambda x.a[e], s) & = & \{\ell \mid \text{there exists } v_1, s_1, v_2, s_2 \text{ such that} \\
 & & \qquad C_{app}(v_1, s_1) \text{ and } s_1(\ell) = s(\ell) \text{ for all } \ell \in L_{env}\text{, and either} \\
 & & \qquad\quad \{\ell \leftarrow \emptyset\}, e\{x \leftarrow v_1\}, s_1 \vdash a \rightarrow \texttt{err}\text{, or} \\
 & & \qquad\quad \emptyset, e\{x \leftarrow v_1\}, s \vdash a \rightarrow v_2/s_2 \text{ and } \ell \in ML(v_2, s_2) \cup ML(e_{api}, s_2)\} \\
ML(e, s) & = & \displaystyle\bigcup_{x \in \mathrm{Dom}(e)} ML(e(x), s)
\end{array}
$$

Figure 4: Definition of the set $ML(v, s)$ of locations modifiable from value v in store s

From the technical side, requiring that no `ref` types occur in the applet's interface E_{api} has an interesting consequence: only source terms from the browser can operate directly on locations allocated by the browser, and only terms from the applet can operate directly on locations allocated by the applet. Thus, all references are contained (in the sense of appendix A, taking T to be the set of all types where the `ref` constructor occurs) either in the browser or in the applet, but never go from one world to the other.

We then rely on the notion of containment to define the set $ML(v, s)$ of locations modifiable from value v in store s as the smallest fixpoint of the equations shown in figure 4. The case for closures follows conditions 1 and 4 in the informal discussion from section 3.2. For condition 5, we use the condition $C_{app}(v_1, s_1)$ instead of the more natural $\ell \notin ML(v_1, s_1)$, so that the equations remain increasing in ML and the existence of the smallest fixpoint is guaranteed. The typing hypothesis (that E_{api} contains no `ref` types) renders condition 3 vacuous, and also dispenses us with defining ML over locations.

The following lemma show that modifiable locations are indeed the only locations modified during the application of a closure.

Proposition 8 *Let p be a set of locations, with $p \subseteq L_{env}$. Let $\lambda x.a_{env}[e]$ be a closure of an environment function. Assume $p \cap ML(\lambda x.a[e], s) = \emptyset$ and $C_{app}(v, s)$. If $Prot(p), e\{x \leftarrow v\}, s \vdash a \rightarrow r$, then $r \neq$ `err`. Instead, $r = v'/s'$, and moreover $p \cap ML(v', s') = \emptyset$ and $p \cap ML(e_{api}, s') = \emptyset$.*

Proof: Assume, by way of contradiction, that $r =$ `err`. Given the evaluation rules, there must exist $\ell \in p$ such that $\{\ell \leftarrow \emptyset\}, e\{x \leftarrow v\}, s \vdash a \rightarrow$ `err`. By definition of ML, this means that $\ell \in ML(\lambda x.a[e], s)$. This contradicts the hypothesis $p \cap ML(\lambda x.a[e], s) = \emptyset$. Hence, $r \neq$ `err`. We therefore have $Prot(p), e\{x \leftarrow v\}, s \vdash a \rightarrow v'/s'$ for some v' and s'. Given the evaluation rules, this implies that we can also derive $\emptyset, e\{x \leftarrow v\}, s \vdash a \rightarrow v'/s'$. Hence, by definition of ML, any ℓ belonging to $ML(v', s')$ or $ML(e_{api}, s')$ also belongs to $ML(\lambda x.a[e], s)$. It follows that $p \cap ML(v', s') = \emptyset$ and $p \cap ML(e_{api}, s') = \emptyset$. □

Using the notion of modifiable locations instead of reachable locations, we finally obtain a security property similar to property 1: the execution of an applet cannot write to locations that are not modifiable from the initial execution environment.

Security property 5 *Assume $S \models e_{api} : E_{api}$ and $\models s : S$. Further assume that E_{api} contains no `ref` types. If $p \subseteq \mathrm{Dom}(s)$ and $p \cap ML(e_{api}, s) = \emptyset$, then for all applets a, we have $Prot(p), e_{api}, s \vdash a \not\rightarrow$ `err`.*

The property follows from the inductive proposition below.

Proposition 9 *Assume $S \models e_{api} : E_{api}$ and $\models s : S$ and E_{api} contains no `ref` types. Further assume $p \subseteq L_{env}$ and $p \cap ML(e_{api}, s) = \emptyset$. Assume $E \vdash a_{app} : \tau$ and $S \models e : E$ and $C_{app}(e, s)$. If $Prot(p), e, s \vdash a_{app} \rightarrow r$, then $r \neq$ `err`. Instead, $r = v'/s'$ and we have $p \cap ML(e_{api}, s') = \emptyset$*

Proof: The proof is by induction on the evaluation derivation. We rely on propositions 2 and 7 to ensure that the semantic typing and containment hypotheses go through the induction. The two non-obvious cases are assignment and application of a closure of a browser function. We write $\varphi = Prot(p)$.

Assignment: a is $a_1 := a_2$. Applying the induction hypothesis twice, we obtain $\varphi, e, s \vdash a_1 \rightarrow v_1/s_1$ and $\varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2$, with $C_{app}(v_1, s_2)$ and $C_{app}(v_2, s_2)$ and $S_2 \models v_1 : \sigma$ `ref` and $S_2 \models v_2 : \sigma$ and $p \cap ML(e_{api}, s_2) = \emptyset$. Hence, v_1 is a location ℓ, and since v_1 is contained in *app*, we have $\ell \in L_{app}$. Thus, $\ell \notin p$ and the evaluation of the assignment does not result in `err`. Moreover, by definition of ML, we have $ML(e_{api}, s) = ML(e_{api}, s')$ if $s(\ell) = s'(l)$ for all $\ell \in L_{env}$. In the present case, the store s' at the end of the evaluation is $s_2\{\ell \leftarrow v_2\}$, with $\ell \notin L_{env}$ by containment, hence $ML(e_{api}, s') = ML(e_{api}, s_2)$ and thus $p \cap ML(e_{api}, s') = \emptyset$ as expected.

Application: a is $a_1(a_2)$. By induction hypothesis, we have $\varphi, e, s \vdash a_1 \rightarrow v_1/s_1$ and $\varphi, e, s_1 \vdash a_2 \rightarrow v_2/s_2$, with $C_{app}(v_1, s_2)$ and $C_{app}(v_2, s_2)$ and $S_2 \models v_1 : \sigma \rightarrow \tau$ and $S_2 \models v_2 : \sigma$ and $p \cap ML(e_{api}, s_2) = \emptyset$. Hence, v_1 is a closure $\lambda x.a'_m[e']$. If $m = app$ (the function comes from the applet), the result follows from the induction hypothesis applied to the evaluation of a'. If $m = env$ (the function comes from the applet environment), the closure $\lambda x.a'_m[e']$ can only be obtained by looking up a variable bound in e_{api}, then possibly performing some function applications or `fst` and `snd` operations. Thus, we have $ML(\lambda x.a'_m[e'], s_2) \subseteq ML(e_{api}, s_2)$, and the result follows by proposition 8. □