

Symbolic Abstract Heaps for Polymorphic Information-flow Guard Inference (Extended Version)

Nicolas Berthier¹ and Narges Khakpour²

¹ OCamlPro (France), University of Liverpool (UK)

² Newcastle University (UK), Linnæus University (Sweden)

Abstract. In the realm of sound object-oriented program analyses for information-flow control, very few approaches adopt flow-sensitive abstractions of the heap that enable a precise modeling of implicit flows. To tackle this challenge, we advance a new symbolic abstraction approach for modeling the heap in Java-like programs. We use a store-less representation that is parameterized with a family of relations among references to offer various levels of precision based on user preferences. This enables us to automatically infer polymorphic information-flow guards for methods via a co-reachability analysis of a symbolic finite-state system. We instantiate the heap abstraction with three different families of relations. We prove the soundness of our approach and compare the precision and scalability obtained with each instantiated heap domain by using the IFSPEC benchmarks and real-life applications.

1 Introduction

Information Flow Control (IFC) mechanisms offer an effective approach to prevent unwanted disclosure of confidential information, or illegal tampering of data. Their task is to ensure confidentiality and/or integrity, which are usually formalized as noninterference baseline properties [1]. Confidentiality demands that *high-sensitive* (secret) inputs do not influence *low-sensitive* (public) outputs. This means that any change in the value of a secret input must not induce a change in any public output. In other words, there must be no *information flow* from any secret to any public output. In software programs, information may flow *explicitly* via direct assignments, e.g., from y to x in $x = y$;, or *implicitly* when the execution of statements is guarded by a condition, e.g., from c to x in **if** ($c > 0$) $x = 42$;

Many static analysis approaches to ensure noninterference have been advanced, that rely on type-systems [2–6], self-composition [7–9], theorem-proving [10], and abstract interpretation [11–14]. *Flow-insensitive* static analyses (with “flow” as in “control-flow”) deal with a single set of facts that is valid for all possible executions of the whole program, whereas *flow-sensitive* analyses provide one set of facts for each statement. In general, flow-sensitivity increases precision, yet comes with an additional computational cost. Heap abstractions as computed

by alias or points-to analyses obey the same principle [15]: a flow-insensitive heap analysis provides a single, finite representation of the conceptually infinite set of memory locations manipulated by all entire executions of the program, while a flow-sensitive variant gives an abstraction of the heap at each statement. Note that a flow-sensitive analysis for object-oriented programs may rely on a flow-insensitive heap abstraction; this means that the analysis must remain imprecise when dealing with heap-allocated structures.

We consider a **Java**-style low-level object-oriented language whose syntax is close to **Jimple**'s [16], and design a static IFC analysis that *automatically* decides whether a program P implemented in this language *is secure*, i.e., P satisfies a desired noninterference property. Several approaches have been proposed to verify such properties for **Java**-style languages [4, 5, 10, 14, 17–21]. To the best of our knowledge, however, none of the sound and scalable solutions rely on a flow-sensitive heap abstraction. Our analysis is therefore *the first of its kind*, as it is *sound*, shows potential for *scalability* since it supports *modularity*, and both: (i) *captures implicit flows*; and (ii) *relies on a flow-sensitive heap abstraction*. Achieving these goals in combination is challenging because the analysis must track the information flows that result from manipulations of object fields and references performed in the program branches that are taken *as well as* in any program branch that is *not* taken: it must therefore reflect about the states of the heap in both taken and non-taken branches simultaneously.

To do so, we use a *security typing environment* that associates each memory location manipulated by the program P with a *security level*. In the case of confidentiality, such a level indicates whether the memory location may hold high-sensitive (secret) data, and P is secure if no value from a high-sensitive *source* flows to a *sink* statement. To deal with all information flows in the heap, we introduce the notion of *symbolic abstract heap domains*, that combine a flow-sensitive security typing environment for every object reachable from a given set of reference variables R , with a flow-sensitive representation of a set of *heap-related relations* pertaining to R (e.g., aliasing). The domains are parametric in a *family of heap-related relations*, which defines the relations that are captured flow-sensitively by the domain. This allows us to define multiple heap abstractions, each one with its level of precision. Abstract heaps in such a domain are *predicates* in a propositional logic, that provide a *store-less* model of the heap where irrelevant details related to the behaviors of the memory allocator and garbage collection are safely abstracted away. The semantics of reference and object mutations are specified using *predicate transformers*. These can be used to encode the security semantics of any method m of the program by means of a symbolic transition system S_m , where the desired noninterference property is reduced to a safety property φ_m [22].

Artifacts that we can infer for a method m include an *information-flow guard*, that is a predicate expressed on propositional “facts” about security levels and heap-related relations pertaining to m 's formal arguments. This guard describes (sufficient) circumstances upon which m is secure, and it is *polymorphic* since it is valid in *any calling context*. More elaborate artifacts that additionally describe

the *effects* of m on the heap enable *sound* inter-procedural analyses. In the present paper, however, we focus on our approach for abstracting the heap and concentrate our exposition on the inference of guards; we leave the inference of polymorphic effects for a future publication. We compute the guard for a method m via a co-reachability analysis of the system S_m w.r.t. its safety property φ_m . A co-reachability analysis finds all states from which a given set of states may be reached, and is typically solved using a fixed-point [23, 24]. We have implemented the guard inference algorithm in a prototype tool called **Guardies**, available at <http://nberth.space/symmaries>, that is equipped with several instantiations of symbolic abstract heap domains using various families of heap-related relations.

Summary of Contributions

- We introduce a novel notion of symbolic abstract heap domains that uses a set of relations to represent the heap, and is the first flow-sensitive heap model used for information-flow control analysis. We define three different instances of this domain (**deep**, **shal**, **dumb**), each with a different set of relations (Section 3), and show that **deep** constitutes a secure heap abstraction (Section 4);
- We symbolically specify the security semantics of our input language to capture explicit and *implicit* flows via the heap, and infer polymorphic information-flow guards via a co-reachability analysis (Section 5). We prove that our analysis under a secure heap abstraction guarantees termination-insensitive noninterference [3];
- We empirically study the respective impacts of our three heap domains, in terms of precision on IFSPEC benchmarks [25], and in terms of scalability with 60 real-life ABM applications [26] (Section 7). Our experiments show that our approach offers the best precision, and the heap model precision has an inverse relationship with scalability. The heap domains **dumb** and **deep**, that are resp. the least- and most-precise heap model, improve the state-of-the-art precision by 2.4% and 4.2% respectively.
- While the existing approaches use an *ad hoc* (flow-insensitive) heap model, **Guardies** offers six different heap models, thereby allowing the user to choose a suitable heap model based on her preferences for precision and scalability.

2 Preliminaries

Input Programs We consider a Java-style low-level language where the code of a method is a non-empty finite semicolon-separated sequence of statements built according to \mathbb{S} in the grammar on the right, where square brackets denote optional constructs. \checkmark is an empty sequence of statements, and $lbl \in Labels$ is a label that uniquely identifies a statement. c is a class name, f_p (resp. f_r) is a primitive (resp. reference) field name, p is a scalar constant, and e is an expression. v (resp. r) depicts any local primitive (resp. reference)

$$\begin{aligned}
 \mathbb{S} &::= [lbl:] a; \mathbb{S} \mid \checkmark \\
 a &::= v = e \mid v = r.f_p \mid r.f_p = e \\
 &\quad \mid r = r \mid r = r.f_r \mid r.f_r = r \\
 &\quad \mid r = \mathbf{new} \ c \mid r = \mathbf{null} \mid r.m(w) \\
 &\quad \mid \mathbf{goto} \ lbl \mid \mathbf{if} \ (e) \ \mathbf{goto} \ lbl \\
 &\quad \mid \mathbf{output}_l(v) \mid \mathbf{output}_l(r) \\
 e &::= p \mid v \mid \ominus e \mid e \oplus e \mid r == r
 \end{aligned}$$

variable or formal argument used of the method in \mathbb{S} . $\text{output}_l(v)$ sends data v over a channel with the security label l . The label l belongs to a *two-level security domain* which is formalized as a lattice $(\mathbb{L}, \sqsubseteq, \sqcup)$, where $\mathbb{L} = \{\perp, \top\}$ is the set of security levels, \perp is the low-sensitive label, \top is the high-sensitive label, \sqsubseteq is a partial order defined over \mathbb{L} with $\perp \sqsubseteq \top$, and \sqcup gives the least-upper-bound³. Information may become public via *sink* statements, that we denote $\text{output}_\perp(x)$. Therefore, $\text{output}_\perp(v)$ is a sink for the value of v , and $\text{output}_\perp(r)$ is a sink for every object that is reachable in the heap via the reference r .

Symbolic Control-flow Graphs — SCFGs The transition systems that we use to encode the security semantics are traditional labeled transition systems augmented with sets of *state* and *input* variables, respectively denoted X and I . The values for input variables can be seen as coming from the environment of the system.

Definition 1 (Symbolic Control-flow Graph). *A symbolic control-flow graph is a tuple $S = \langle A, X, I, \Delta, \ell_0, X_0 \rangle$ where: A is a finite non-empty set of locations; X and I respectively denote state and input variables; Δ is a set of transitions labeled with a guard that is a predicate on state or input variables, and a possibly empty set of assignments to state variables, noted $[v_0 := e_0, \dots, v_n := e_n]$, or \emptyset if empty; $\ell_0 \in A$ is the initial location; and X_0 is a predicate that describes the entire set of possible initial valuations for all the state variables.*

Predicates and right-hand-side expressions in assignments are built using traditional logical connectives (i.e., \neg , \vee , \wedge and \Rightarrow), along with a ternary conditional construct “if · then · else.” with an obvious meaning. The symbolic variables we make use of typically take their values in the security domain \mathbb{L} , or the set of Booleans $\mathbb{B} \stackrel{\text{def}}{=} \{\text{ff}, \text{tt}\}$. We use a *merge operation* \sqcup to merge variable assignments. This operation is obtained as a union where multiple expressions assigned to a variable v are combined using some connective \sqcup_v . The latter depends on the semantics of each variable: as we only use variables to hold over-approximations in our encoding, we use the disjunction \vee for Booleans, and the least-upper-bound \sqcup for security levels. For instance, $\{a := \text{tt}, b := \text{ff}\} \sqcup \{b := \text{tt}\} = \{a := \text{tt}, b := \text{ff} \vee \text{tt}\}$. An *invariant* φ for the SCFG S is a mapping from locations to predicates on state variables. S *satisfies* φ iff every state q with location ℓ that is reachable by S is such that $q \models \varphi(\ell)$.

An SCFG S induces a model $\llbracket S \rrbracket$ that is a finite-state automaton whose *state-space* \mathcal{Q}_S is the Cartesian product of the set of locations A and the set of all possible valuations for the state variables, i.e., $\mathcal{Q}_S = A \times \text{Val}(X)$, where $\text{Val}(X)$ is the set of *valuations* for all variables in X . $\llbracket S \rrbracket$ takes one transition whenever it receives a valuation for *all* the input variables, i.e., an element in $\text{Val}(I)$ (an empty set for \mathcal{S}). In any location, there is always exactly one transition whose guard is satisfied by the valuations for all the variables. When this transition is

³ As is traditional, we will present our work by focusing on a standard two-level lattice $\mathbb{L} \stackrel{\text{def}}{=} \{\perp, \top\}$; minor adaptations would be necessary to support more complex lattices.

taken, its assignments are applied to update the state variables. In this work, we only construct SCFGs that are both *deterministic* and *reactive*: i.e., given any location $\ell \in \Lambda$ and valuations for input and state variables, there always exists a unique transition in $\Delta(\ell)$ whose guard is satisfied. The only source of non-determinism that we make use of in the models lies in the sets of initial values for the state variables. Note there is also no notion of accepting state; combined with the properties above, this means that every infinite sequence of elements in $\text{Val}(I)$ leads to a valid run.

3 Symbolic Abstract Heap Domains

We first detail our design of heap abstractions that are suitable for the symbolic encoding of security semantics. In this approach, one predicate is used to model *a set of symbolic heaps*. Each symbolic heap represents a *parameterizable* set of *heap-related relations* between the portions of the heap that are reachable via a given set of references R , along with a *security typing environment* for every reachable portion of heap. Such predicates provide *storeless representations* since object locations are not explicitly represented. Predicate transformers describe the *effects of heap and reference variable mutations* on sets of symbolic heaps.

3.1 Families of Heap-related Relations

Our definition of symbolic abstract heap domains is parameterized by a *family of heap-related relations*. A typical example of a heap-related relation is the aliasing relation, that we denote with the symbol \sim , and which is defined as an equivalence relation where $r \sim s$ holds iff r and s point to the same object. We define a family of heap-related relations as a pair $\text{hd} \stackrel{\text{def}}{=} \langle \mathcal{R}_{\text{Sen}}, \mathcal{R}_{\text{Insen}} \rangle$ where \mathcal{R}_{Sen} is a set of *flow-sensitive relations*, and $\mathcal{R}_{\text{Insen}}$ are *constant flow-insensitive relations* (or facts). A relation is formally specified as a set of Boolean variables that each indicates whether two references taken from R are in the relation or not (i.e., we use predicate abstraction where a Boolean variable specifies whether a relation between two references holds). For instance, we need four propositions (therefore, as many Boolean variables) to represent the relation \frown defined over $R = \{a, b\}$, i.e., $a \frown a$, $a \frown b$, $b \frown a$ and $b \frown b$. The proposition $x \frown y$ evaluates to true if x is in the relation \frown with y . Further, a relation $\frown = \{(a, a), (b, a)\}$ is formalized as $a \frown a \wedge \neg a \frown b \wedge b \frown a \wedge \neg b \frown b$.

The propositions about flow-sensitive relations may be updated by the program statements, while the propositions about constant flow-insensitive relations are straightforwardly substituted with `tt` or `ff`, and serve the sole purpose of improving the precision of the predicate transformers that manipulate symbolic abstract heaps. For instance, in a heap domain that does not handle the aliasing relation flow-sensitively, two references of incompatible types can never alias each other. We formalize this with the pre-analysis function `CanRelate`, that returns three-valued *sound* facts about the relations in $\mathcal{R} = \mathcal{R}_{\text{Insen}} \cup \mathcal{R}_{\text{Sen}}$ w.r.t. the set of reference variables R . Given any relation $\frown \in \mathcal{R}$ and a pair of references $(r, s) \in R^2$,

$\text{CanRelate}(r \frown s)$ returns **yes** if $r \frown s$ always holds, **no** if it cannot hold, or **maybe** otherwise. In its most trivial form, this pre-analysis function operates on a purely lexical level, e.g., by returning **yes** if queried for $r \frown r$ with \frown a reflexive relation, **maybe** otherwise. It may additionally involve an analysis of the class hierarchy and take the declared type of the reference variables into account to give more precise facts. Note that CanRelate helps us simplify the heap formulae by reducing the number of propositions used to represent heaps. For instance, if \frown is a reflexive relation in our previous example, we don't need to consider the propositions $a \frown a$ and $b \frown b$, and use the constant tt instead. We leave further specifications of the pre-analysis open for the sake of modularity.

We need to differentiate between “flow-sensitive heaps” and “flow-sensitive heap relations”. The first case means that the heap changes during the execution while the latter states that the relation used to specify the heap structure changes during the execution (i.e., \mathcal{R}_{Sen}). Therefore, a flow-sensitive symbolic heap abstraction models at least one heap relation flow-sensitively.

3.2 Symbolic Abstract Heap Domain

Formally, a *symbolic abstract heap domain* for the family of heap-related relations hd is defined as a pair $\text{HeapDom}_{\text{hd}} \stackrel{\text{def}}{=} \langle \mathbb{H}_{\text{hd}}, \mathbb{T}_{\text{hd}} \rangle$ where \mathbb{H}_{hd} is the set of symbolic abstract heaps, and \mathbb{T}_{hd} is a set of predicate transformers to manipulate the abstract heaps. A *symbolic abstract heap* from this domain is a predicate $\mathfrak{h} \in \mathbb{H}_{\text{hd}}$ defined on two sets of state variables $\mathbb{V}_{\vec{l}}$ and $\mathbb{V}_{\mathcal{R}}$. The set $\mathbb{V}_{\vec{l}}$ associates a *security level variable* $\vec{r}^{\mathfrak{h}}$ with each reference $r \in R$, that represents an *upper bound* on the security levels of any object that is reachable via r : these variables constitute the *security typing environment* for the abstract heap \mathfrak{h} . In turn, the set $\mathbb{V}_{\mathcal{R}}$ consists of Boolean variables that describe *over-approximations of flow-sensitive heap-related relations* between the references in R , i.e., a variable $r \frown^{\mathfrak{h}} s \in \mathbb{V}_{\mathcal{R}}$ holds whenever (r, s) *may* be in the heap-related relation \frown :

$$\mathbb{V}_{\mathcal{R}} \stackrel{\text{def}}{=} \{ r \frown^{\mathfrak{h}} s \mid (r, s) \in R^2, \frown \in \mathcal{R}_{\text{Sen}}, \text{CanRelate}(r \frown s) = \text{maybe} \}.$$

Further, \mathbb{V}_{ff} and \mathbb{V}_{tt} are sets of constants that capture all relations that never hold and always hold according to the function CanRelate , respectively. (For the sake of readability, we will omit the exponent \mathfrak{h} of security-level variables when a single abstract heap \mathfrak{h} is involved, i.e., $\vec{r}^{\mathfrak{h}}$ will be denoted \vec{r} .)

$$\begin{aligned} \mathbb{V}_{\text{ff}} &\stackrel{\text{def}}{=} \{ r \frown^{\mathfrak{h}} s \mid (r, s) \in R^2, \frown \in \mathcal{R}_{\text{Sen}} \cup \mathcal{R}_{\text{Insen}}, \text{CanRelate}(r \frown s) = \text{no} \} \\ \mathbb{V}_{\text{tt}} &\stackrel{\text{def}}{=} \left\{ r \frown^{\mathfrak{h}} s \mid (r, s) \in R^2, \left(\frown \in \mathcal{R}_{\text{Sen}}, \text{CanRelate}(r \frown s) = \text{yes} \right) \vee \left(\frown \in \mathcal{R}_{\text{Insen}}, \text{CanRelate}(r \frown s) \neq \text{no} \right) \right\}. \end{aligned}$$

We summarize in Table 1 the main denotations that we use to represent and manipulate symbolic abstract heaps. The leftmost column shows the variables that represent security levels and heap-related relations, along with the operator ($\text{null } R'$). The right-hand side column lists the set of *predicate transformers* that can be applied on a symbolic abstract heap to alter its representation. The two

Table 1: Denotations for symbolic abstract heaps

Two symbolic abstract heaps:	$\mathfrak{h}, \mathfrak{h}' \in \mathbb{H}_{\text{hd}}$
Set of variables encoding a heap-related relation $\wedge \in \mathcal{R}_{\text{Sen}}$:	$\mathbb{V}_{\mathcal{R}}$
Set of constants encoding non-membership facts, for any relation $\wedge \in \mathcal{R}_{\text{Sen}} \cup \mathcal{R}_{\text{Insen}}$:	\mathbb{V}_{ff}
Set of constants encoding membership facts, for any relation $\wedge \in \mathcal{R}_{\text{Sen}} \cup \mathcal{R}_{\text{Insen}}$:	\mathbb{V}_{tt}
Set of variables encoding the security levels:	$\mathbb{V}_{\vec{l}}$
Variables & Predicate	Transformers (\mathbb{T}_R)
Security level variable: $\vec{r}^{\mathfrak{h}}$ (or \vec{r})	Reference assignment: (as)
Relation variable: $r \wedge^{\mathfrak{h}} s$	Mutation and allocation: $(mu, \uparrow l)$
Initialization: $(\text{null } R')$	Bulk upgrade: $\text{BulkUpgr}_{\mathfrak{h} \leftarrow \mathfrak{h}'}$

with $as \in \{r = s, r = s.f_r, r = \text{null}\}$, $mu \in \{r.f_p \rightsquigarrow, r.f_r = s, r = \text{new}\}$, $(s, r) \in R^2$, $R' \subseteq R$, and any security level expression l .

```

1 class A { int fi; } class B { A fa; }
2 static void m (A a, B b, int i) {
3   B r = new B;
4   a.fi = i;
5   r.fa = a;
6   output $\perp$ (b);
7 }

```

(a) Class definitions and method m .

Statement Loc	Heap Transformer
3	$(r = \text{new}, \uparrow pc)$
4	$(a.fi \leftarrow, \uparrow \vec{i} \sqcup pc)$
5	$(r.fa = a, \uparrow \vec{a} \sqcup \vec{a} \sqcup pc)$
6	\emptyset

$$R = \{a, b, r\}, \mathcal{R} = \{\sim, \overset{*}{\leftarrow}\}$$

$$\mathbb{V}_{\vec{l}} = \{\vec{a}, \vec{b}, \vec{r}\}, \mathbb{V}_{\mathcal{R}} = \{b \rightsquigarrow r, b \overset{*}{\leftarrow} a, r \overset{*}{\leftarrow} a\}$$

(b) Heap Transformers and Variables

Fig. 1: Method that manipulates references, with a representation of heap transformers, heap-related relationships and variables.

first transformers in the column operate in accordance with a given reference assignment (as) or mutation (mu). The expression l given to the latter gives the security level of the information that flows to mutated objects. We give in Fig. 2 the definitions of all transformers. These definitions make use of functions specialized for each family of heap-related relations, detailed below. $(\text{null } R')$ builds a predicate that constrains variables in $\mathbb{V}_{\vec{l}}$ and $\mathbb{V}_{\mathcal{R}}$ to account for the fact that a given set of references $R' \subseteq R$ is **null**—this notably entails that every object reachable via R' is low-sensitive. The *bulk upgrade* is a transformer used to capture implicit flows through the heap by joining two distinct heap abstractions \mathfrak{h} and \mathfrak{h}' that belong to the same domain. More specifically, this transformer assumes that $\vec{r}^{\mathfrak{h}'} \sqsubseteq \vec{r}^{\mathfrak{h}}$ for all $r \in R$, and: (i) copies the heap-related relations from \mathfrak{h}' to \mathfrak{h} ; and (ii) upgrades the typing environment of \mathfrak{h} via a pairwise join with the corresponding levels in \mathfrak{h}' .

Example 1. Consider method m given in Fig. 1(a). Fig. 1(b) shows its references R , heap-related relations \mathcal{R} , references security levels $\mathbb{V}_{\vec{l}}$ and variables $\mathbb{V}_{\mathcal{R}}$ to specify the heap structure using relations \mathcal{R}_{Sen} . Further, the table in this Figure shows the heap transformers associated with each statement to update the heap relations and reference security levels. As an example transformer, consider $(r = \text{new}, \uparrow pc)$ that corresponds to the statement $r = \text{new } B$; With a heap domain that captures the aliasing relation flow-sensitively, the resulting set of assignments includes (at least) $[b \rightsquigarrow r := \text{ff}, \vec{r} := l]$ where l is a security level expression $s.t. l \sqsupseteq pc$.

$$\begin{aligned}
(r = \text{null}) &\stackrel{\text{def}}{=} \text{UpdHpRel}(r = _) \quad \Downarrow [\vec{r} := \perp] \\
(r = s) &\stackrel{\text{def}}{=} \text{UpdHpRel}(r = s) \quad \Downarrow [\vec{r} := \vec{s}] \\
(r = s.f_r) &\stackrel{\text{def}}{=} \text{UpdHpRel}(r = s.f_r) \quad \Downarrow [\vec{r} := \vec{s}] \\
(r = \text{new}, \uparrow l) &\stackrel{\text{def}}{=} \text{UpdHpRel}(r = _) \quad \Downarrow [\vec{r} := l] \\
(r.f_p \leftarrow, \uparrow l) &\stackrel{\text{def}}{=} \text{UpdHpLev}(r, l) \\
(r.f_r = s, \uparrow l) &\stackrel{\text{def}}{=} \text{UpdHpRel}(r.f_r = s) \quad \Downarrow \text{UpdHpLev}(r, l) \\
(\text{null } R') &\stackrel{\text{def}}{=} \text{NullRefs}_{\mathfrak{h}}(R') \quad \wedge \bigwedge_{r \in R'} \vec{r} = \perp \\
\text{BulkUpgr}_{\mathfrak{h} \leftarrow \mathfrak{h}'} &\stackrel{\text{def}}{=} \text{CopyRels}_{\mathfrak{h} \leftarrow \mathfrak{h}'} \quad \Downarrow \text{RstrLev}_{\mathfrak{h} \leftarrow \mathfrak{h}'} \\
\text{with } \text{CopyRels}_{\mathfrak{h} \leftarrow \mathfrak{h}'} &\stackrel{\text{def}}{=} \bigsqcup_{r \sim^{\mathfrak{h}'} s \in \mathcal{V}_{\mathcal{R}}} [r \sim^{\mathfrak{h}} s := r \sim^{\mathfrak{h}'} s] \quad \text{and } \text{NullRefs}_{\mathfrak{h}}(R') \stackrel{\text{def}}{=} \bigwedge_{r \sim^{\mathfrak{h}} s \in \mathcal{V}_{\mathcal{R}}, \{r, s\} \cap R' \neq \emptyset} (r \sim^{\mathfrak{h}} s = \text{ff}).
\end{aligned}$$

Fig. 2: Definitions of generic transformers \mathbb{T}_R for any symbolic abstract heap domain $\text{HeapDom}_{\text{hd}}$. Note \mathfrak{h} and \mathfrak{h}' belong to the same domain, i.e., $(\mathfrak{h}, \mathfrak{h}') \in \mathbb{H}_R^2$, and $R' \subseteq R$. See Fig. 3 for an example definition of $\text{UpdHpRel}(\cdot)$.

3.3 Instances of Symbolic Abstract Heap Domains

We present three instances of the domain introduced above. We first define the *transitive “field-aliasing”* relation, denoted with the symbol $\overset{*}{\leftarrow}$, which states for any given pair of references r and s , $r \overset{*}{\leftarrow} s$ holds whenever a reference field of an object reachable via r is an alias of s . This relation allows heap domains to capture some useful facts about the structure of the graph of objects when it comes to maintaining object types such as security levels.

We now assume a sound pre-analysis function CanRelate over $\{\sim, \overset{*}{\leftarrow}\}$, and use the above relations to define the three families of heap-related relations based on which we shall instantiate our symbolic abstract heap domains:

- $\text{HeapDom}_{\text{deep}}$, with $\text{deep} \stackrel{\text{def}}{=} \langle \{\sim, \overset{*}{\leftarrow}\}, \emptyset \rangle$, uses symbolic variables to represent over-approximations of aliasing and field-aliasing relations;
- $\text{HeapDom}_{\text{shal}}$, with $\text{shal} \stackrel{\text{def}}{=} \langle \{\sim\}, \{\overset{*}{\leftarrow}\} \rangle$, only maintains a flow-sensitive over-approximation of the aliasing relation, yet makes use of field-aliasing facts to improve the precision of transformers;
- $\text{HeapDom}_{\text{dumb}}$, with $\text{dumb} \stackrel{\text{def}}{=} \langle \emptyset, \{\sim, \overset{*}{\leftarrow}\} \rangle$, does not represent any flow-sensitive heap-related relation, yet makes use of flow-insensitive (field-)aliasing relations.

Table 2: Variables and constants involved in representing \mathfrak{h} when analyzing \mathfrak{m} , for each domain. \mathfrak{h} exponents have been omitted for readability.

	deep	shal	dumb
$\mathcal{V}_{\vec{r}}$		$\{\vec{a}, \vec{b}, \vec{r}\}$	
$\mathcal{V}_{\mathcal{R}}$	$\{\mathfrak{b} \sim \mathfrak{r}, \mathfrak{b} \overset{*}{\leftarrow} \mathfrak{a}, \mathfrak{r} \overset{*}{\leftarrow} \mathfrak{a}\}$	$\{\mathfrak{b} \sim \mathfrak{r}\}$	\emptyset
\mathcal{V}_{ff}	$\{\mathfrak{a} \sim \mathfrak{b}, \mathfrak{a} \sim \mathfrak{r}, \mathfrak{a} \overset{*}{\leftarrow} \mathfrak{a}, \mathfrak{a} \overset{*}{\leftarrow} \mathfrak{b}, \mathfrak{a} \overset{*}{\leftarrow} \mathfrak{r}, \mathfrak{b} \overset{*}{\leftarrow} \mathfrak{b}, \mathfrak{b} \overset{*}{\leftarrow} \mathfrak{r}, \mathfrak{r} \overset{*}{\leftarrow} \mathfrak{b}, \mathfrak{r} \overset{*}{\leftarrow} \mathfrak{r}\}$		
\mathcal{V}_{tt}	$\{\mathfrak{a} \sim \mathfrak{a}, \mathfrak{b} \sim \mathfrak{b}, \mathfrak{r} \sim \mathfrak{r}\}$	$\{\mathfrak{a} \sim \mathfrak{a}, \mathfrak{b} \sim \mathfrak{b}, \mathfrak{r} \sim \mathfrak{r}, \mathfrak{b} \overset{*}{\leftarrow} \mathfrak{a}, \mathfrak{r} \overset{*}{\leftarrow} \mathfrak{a}\}$	$\{\mathfrak{a} \sim \mathfrak{a}, \mathfrak{b} \sim \mathfrak{b}, \mathfrak{r} \sim \mathfrak{r}, \mathfrak{b} \overset{*}{\leftarrow} \mathfrak{a}, \mathfrak{r} \overset{*}{\leftarrow} \mathfrak{a}\}$

$$\begin{aligned}
\text{UpdHpRel}(r = _) &\stackrel{\text{def}}{=} \bigsqcup_{d \in R} \left[d \sim r := \text{ff}, r \overset{*}{\hookrightarrow} d := \text{ff}, d \overset{*}{\hookrightarrow} r := \text{ff} \right] \\
\text{UpdHpRel}(r = s) &\stackrel{\text{def}}{=} \bigsqcup_{d \in R} \left[d \sim r := d \sim s, r \overset{*}{\hookrightarrow} d := s \overset{*}{\hookrightarrow} d, d \overset{*}{\hookrightarrow} r := d \overset{*}{\hookrightarrow} s \right] \\
\text{UpdHpRel}(r = s.f_r) &\stackrel{\text{def}}{=} \bigsqcup_{d \in R} \left[d \sim r := s \overset{*}{\hookrightarrow} d, r \overset{*}{\hookrightarrow} d := s \overset{*}{\hookrightarrow} d, d \overset{*}{\hookrightarrow} r :=, d \sim s \vee d \overset{*}{\hookrightarrow} s \right] \\
\text{UpdHpRel}(r.f_r = s) &\stackrel{\text{def}}{=} \bigsqcup_{a \overset{*}{\hookrightarrow} b \in \mathbb{V}_{\mathcal{R}}} \left[a \overset{*}{\hookrightarrow} b := a \overset{*}{\hookrightarrow} b \vee \left((a \sim r \vee a \overset{*}{\hookrightarrow} r) \wedge (b \sim s \vee s \overset{*}{\hookrightarrow} b) \right) \right] \\
\text{UpdHpLev}(r, l) &\stackrel{\text{def}}{=} \bigsqcup_{s \in R} \left[\vec{s} := \vec{s} \sqcup \text{ if } s \sim r \vee s \overset{*}{\hookrightarrow} r \text{ then } l \text{ else } \perp \right] \\
\text{RstrLev}_{\mathfrak{h} \leftarrow \mathfrak{h}'} &\stackrel{\text{def}}{=} \bigsqcup_{(r,s) \in R^2} \left[\vec{s}^{\mathfrak{h}} := \vec{s}^{\mathfrak{h}} \sqcup \text{ if } s \sim^{\mathfrak{h}'} r \vee s \overset{*}{\hookrightarrow}^{\mathfrak{h}'} r \text{ then } \vec{r}^{\mathfrak{h}'} \text{ else } \perp \right]
\end{aligned}$$

Fig. 3: Specialized functions for updating security level and relation variables for each domain defined with $\text{hd} \in \{\text{deep}, \text{shal}, \text{dumb}\}$; \mathfrak{h} exponents have been omitted when a single abstract heap is involved.

Example 2. Consider method \mathbf{m} given in Fig. 1(a), and assume a class hierarchy pre-analysis. We instantiate the three symbolic abstract heap domains as $\text{HeapDom}_{\text{hd}}$ and define an abstract heap $\mathfrak{h} \in \mathbb{H}_{\text{hd}}$, for each $\text{hd} \in \{\text{deep}, \text{shal}, \text{dumb}\}$. Table 2 shows the sets of variables used by each one of these domains to represent \mathfrak{h} ($\mathbb{V}_{\vec{L}}$ and $\mathbb{V}_{\mathcal{R}}$), along with the symbols that denote constants involved in capturing relations flow-insensitively (\mathbb{V}_{ff} and \mathbb{V}_{tt}).

Regarding transformers, we give in Fig. 3 the specialized functions used by their definitions in Fig. 2. The domains defined with $\text{hd} \in \{\text{deep}, \text{shal}, \text{dumb}\}$ share these definitions. $\text{UpdHpRel}(r = _)$ updates relation variables in \mathfrak{h} to reflect the erasing of a given reference r to either nil or a fresh reference by clearing variables from $\mathbb{V}_{\mathcal{R}}$. In turn, $\text{UpdHpRel}(r = s)$ updates the variables in $\mathbb{V}_{\mathcal{R}}$ to encode the copy of a reference s to r . $\text{UpdHpRel}(r = s.f_r)$ makes any reference d that may be an alias of one of s 's field a potential alias of r (when a corresponding variable $d \sim^{\mathfrak{h}} r$ belongs to $\mathbb{V}_{\mathcal{R}}$), and updates any variable that represents the $\overset{*}{\hookrightarrow}$ relation to reflect that s becomes a field-alias of r . $\text{UpdHpRel}(r.f_r = s)$ makes s a field-alias of r while maintaining transitivity of $\overset{*}{\hookrightarrow}$. Storing a reference may only add elements in relation $\overset{*}{\hookrightarrow}$, hence the disjunction in every assignment defined by this operation. Observe that, as can be seen in the definition of $\text{UpdHpRel}(r = s.f_r)$ for $\text{hd} = \text{shal}$, instead of simply setting $d \sim r := \text{tt}$ for every potential alias d of r (i.e., blindly assuming that no information is known about the potential aliasing relation), a constant $s \overset{*}{\hookrightarrow} d \in \mathbb{V}_{\text{tt}} \cup \mathbb{V}_{\text{ff}}$ is used instead to further restrict the new potential aliases to the cases that have not been ruled out by the pre-analysis. The **dumb** domain involves some pre-established facts via a similar mechanism. $\text{UpdHpLev}(r, l)$ takes a reference r and a security level expression l , and upgrades the security level associated with the objects reachable via r as well as that

of every reference s that may transitively field-alias r (i.e., $s \xrightarrow{*} r$). $\text{RstrLev}_{\mathfrak{h} \leftarrow \mathfrak{h}'}$ upgrades the typing environment of \mathfrak{h} according to that of \mathfrak{h}' .

4 Secure Heap Abstraction

To specify the semantics of heap operations performed by a program, we define a *concrete heap domain* that maintains the value of *primitive fields* in addition to *the precise heap-related relations*. A concrete heap domain is defined similarly to that of abstract heap domains introduced in Section 3, *with the difference that the heap maintains the primitive fields instead of security levels*. The concrete heap domain is defined as $\text{HeapDom}_{\text{crt}} = \langle \mathbb{H}_{\text{crt}}, \mathbb{T}_{\text{crt}} \rangle$ where $\text{crt} \stackrel{\text{def}}{=} \langle \{\sim, \xrightarrow{f}\}, \emptyset \rangle$, the relation \sim is an ordinary reference aliasing relation, and \xrightarrow{f} is a field-aliasing relation, i.e., $r \xrightarrow{f} s$ holds iff the field f of the object referenced by r is an alias of s . Fig. 4 presents the operations on the concrete heap domain. The notation $[op]_{\tilde{h}}$ shows the predicate transformer that corresponds to the operation op on a concrete heap $\tilde{h} \in \mathbb{H}_{\text{crt}}$. The functions $\text{UpdHpRel}(op)$ and $\text{UpdPFields}(op)$ respectively specify the updates to the concrete heap-related relations and the primitive fields of the heap \tilde{h} by performing the heap operation op . RFields , default and und show the set of reference fields, the default value and the undefined value, respectively.

Since we use an abstract heap domain to model and analyze information flow via heap, we should ensure that the analysis under abstract heap domains guarantees noninterference. To this end, we first define the notion of *indistinguishable heaps* and then prove that two indistinguishable heaps from the concrete domain remain indistinguishable after applying a heap transformer. Let \mathfrak{h} be a heap from an arbitrary heap domain and $R' \subseteq R$ be a set of references. The reference graph over R' induced by \mathfrak{h} is a labeled digraph $G_{R'}^{\mathfrak{h}} \stackrel{\text{def}}{=} (N_{\mathfrak{h}}, E_{\mathfrak{h}})$ where $N_{\mathfrak{h}} = R'$ is the set of nodes, and the edges $E_{\mathfrak{h}}$ show the heap-related relations between them, i.e., $E_{\mathfrak{h}} = \{(r, \frown, r') \mid \mathfrak{h} \models r \frown r', r \in R' \vee r' \in R'\}$. Let \mathfrak{h} be an abstract heap and $G_{\perp}^{\mathfrak{h}}$ be a sub-graph of $G_R^{\mathfrak{h}}$ containing the low-sensitive references $R_{\perp}^{\mathfrak{h}} = \{r \mid \mathfrak{h} \models (\vec{r} = \perp)\}$. We say two concrete heaps are indistinguishable, if heap-related relations and primitive fields of their low-sensitive references are identical, i.e., (i) the reference graphs corresponding to their low-sensitive portions of the heaps are *isomorphic*, and (ii) the valuation of primitive fields of their low-sensitive references are identical.

Definition 2 (Indistinguishable Heaps). *We say two concrete heaps \tilde{h} and \tilde{h}' from \mathbb{H}_{crt} , are indistinguishable w.r.t. an abstract heap \mathfrak{h} , noted by $\tilde{h} =_{\mathfrak{h}} \tilde{h}'$, iff (i) $G_{\perp}^{\tilde{h}}$ and $G_{\perp}^{\tilde{h}'}$ are isomorphic, denoted by $G_{\perp}^{\tilde{h}} \cong G_{\perp}^{\tilde{h}'}$, and (ii) $\forall x. \tilde{h} \models r.f_p = x \Leftrightarrow \tilde{h}' \models r.f_p = x$, for all $r \in R_{\perp}^{\mathfrak{h}}$ where f_p is a primitive field.*

We define the concept of *secure heap abstraction*, which states that two indistinguishable heaps should remain indistinguishable after applying a heap operation and its corresponding transformer at the abstract heap domain level:

$$[op]_{\mathfrak{h}} \stackrel{\text{def}}{=} \begin{cases} \text{UpdHpRel}(r.f_r = s) & \text{if } op = \langle\langle r.f_r = s \rangle\rangle \\ \text{UpdHpRel}(r = s.f_r) & \text{if } op = \langle\langle r = s.f_r \rangle\rangle \\ \text{UpdHpRel}(r = s) \sqcup \text{UpdPFields}(op) & \text{if } op = \langle\langle r = s \rangle\rangle \\ \text{UpdHpRel}(r = _) \sqcup \text{UpdPFields}(op) & \text{if } op = \langle\langle r = \text{new} \rangle\rangle \\ \text{UpdHpRel}(r = _) \sqcup \text{UpdPFields}(op) & \text{if } op = \langle\langle r = \text{null} \rangle\rangle \\ & [v := s.f_p] \quad \text{if } op = \langle\langle v = s.f_p \rangle\rangle \\ & \text{UpdPFields}(op) \quad \text{if } op = \langle\langle r.f_p = v \rangle\rangle \end{cases}$$

where

$$\begin{aligned} \text{UpdHpRel}(r = _) &\stackrel{\text{def}}{=} \bigsqcup_{r' \in R, f \in \text{RFields}} \left[r \sim r' := \text{ff}, r \xrightarrow{f} r' := \text{ff}, r' \xrightarrow{f} r := \text{ff} \right] \\ \text{UpdHpRel}(r = s) &\stackrel{\text{def}}{=} \bigsqcup_{r' \in R, f \in \text{RFields}} \left[r \sim r' := s \sim r', r \xrightarrow{f} r' := s \xrightarrow{f} r', r' \xrightarrow{f} r := r' \xrightarrow{f} s \right] \\ \text{UpdHpRel}(r = s.f_r) &\stackrel{\text{def}}{=} \bigsqcup_{r' \in R, f' \in \text{RFields}} \left[r \sim r' := s \xrightarrow{f_r} r', r' \xrightarrow{f'} r' := s \xrightarrow{f_r} s' \wedge s' \xrightarrow{f'} r', r' \xrightarrow{f'} r := r' \sim s \right] \\ \text{UpdHpRel}(r.f_r = s) &\stackrel{\text{def}}{=} \bigsqcup_{r' \xrightarrow{f_r} s' \in \mathbb{V}_{\mathcal{R}}} \left[r' \xrightarrow{f_r} s' := r' \sim r \wedge s' \sim s \right] \end{aligned}$$

and

$$\text{UpdPFields}(op) \stackrel{\text{def}}{=} \bigsqcup_{f_p \in \text{PFields}} \begin{cases} [s.f_p := s.f] & \text{if } op = \langle\langle r = s \rangle\rangle \\ [s.f_p := \text{default}] & \text{if } op = \langle\langle r = \text{new} \rangle\rangle \\ [r'.f_p := v] & \text{if } op = \langle\langle r.f_p = v \rangle\rangle, \mathfrak{h} \models r \sim r' \\ [s.f_p := \text{und}] & \text{if } op = \langle\langle r = \text{null} \rangle\rangle \end{cases}$$

Fig. 4: Update of alias variables in a heap \mathfrak{h} from the concrete heap domain.

Definition 3 (Secure Heap Abstraction). *The concrete heap domain $\text{HeapDom}_{\text{crt}}$ is secure w.r.t. an abstract heap domain $\text{HeapDom}_{\text{hd}}$, if and only if it preserves the heap indistinguishability relation, i.e., given any concrete heaps $(\mathfrak{h}_1, \mathfrak{h}_2) \in \mathbb{H}_{\text{crt}} \times \mathbb{H}_{\text{crt}}$, and an abstract heap $\mathfrak{h} \in \mathbb{H}_{\text{hd}}$ s.t $\mathfrak{h}_1 =_{\mathfrak{h}} \mathfrak{h}_2$, it holds that:*

- (a) *for all pair (as, as') of reference assignment statements and their corresponding operations on abstract heaps where $as \in \{\langle\langle r = s \rangle\rangle, \langle\langle r = s.f_r \rangle\rangle, \langle\langle r = \text{null} \rangle\rangle\}$, $\mathfrak{h}'_1 =_{\mathfrak{h}} \mathfrak{h}'_2$ holds where $\mathfrak{h}'_i = [as]_{\mathfrak{h}_i}$, $i \in \{1, 2\}$, and $\mathfrak{h}' = (as')$;*
- (m) *for all pair (mu, mu') of mutation statements and their corresponding operations on heaps where $mu \in \{\langle\langle r.f_p = e \rangle\rangle, \langle\langle r.f_r = s \rangle\rangle, \langle\langle r = \text{new } c \rangle\rangle\}$, for all $l \in \mathbb{L}$ where $\vec{s} \sqsubseteq l$ if mu is $\langle\langle r.f_r = s \rangle\rangle$, and $\vec{e} \sqsubseteq l$ if mu is $\langle\langle r.f_p = e \rangle\rangle$, it holds that $\mathfrak{h}'_1 =_{\mathfrak{h}} \mathfrak{h}'_2$ where $\mathfrak{h}'_i = [mu]_{\mathfrak{h}_i}$, $i \in \{1, 2\}$, and $\mathfrak{h}' = (mu', \uparrow l)$;*

Theorem 1. *The concrete heap domain $\text{HeapDom}_{\text{crt}}$ is secure w.r.t. the deep abstract heap domain $\text{HeapDom}_{\text{deep}}$ according to Definition 3.*

Proof. See Appendix A.

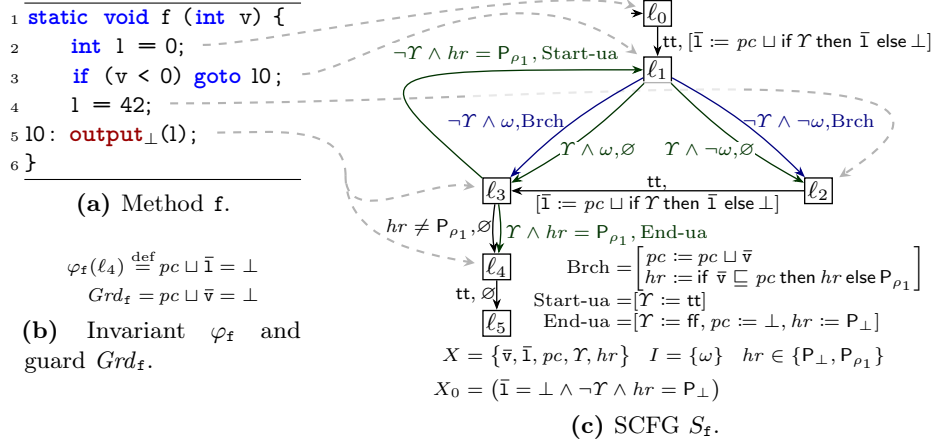


Fig. 5: Example method with SCFG, invariant, and resulting guard.

5 Inferring Polymorphic Information-flow Guards

Let us now put the heap abstraction aside, and focus on our approach for computing guards and capturing implicit flows. We start by considering the method f given in Fig. 5(a), that does not involve any reference variable. f implements a canonical pattern used to illustrate implicit flows: if its argument v is high-sensitive, then the information output on line 5 is also high-sensitive via an implicit flow induced by the assignment guarded by the condition on line 3. For this example, our requirement demands that executing the sink statement on line 5 does not leak confidential information. Therefore, the guard that we want to compute for f is a sufficient condition that allow us to decide whether any call to the method satisfies the confidentiality requirement based on a set of program facts available *whatever the calling context*. For f , the latter set of facts includes, for instance, the security level of the effective argument for v , or whether the call happens in a high context (i.e., if it is guarded by a condition on high-sensitive information). To achieve this, we build the SCFG that specifies the security semantics of any method in such a way that the set of all its potential initial states encodes all the possible calling contexts for the method. For instance, when encoding f the variable \bar{v} assigned to the argument v is left uninitialized (contrary to the level \bar{l} of local variable l). Another state variable that is left uninitialized is pc , as it denotes the security level of the calling context. We then associate a safety property φ that expresses constraints on security levels at states of the SCFG, and use a co-reachability analysis to find the set of *all initial states* from which no run ever leads to a violation of φ . We graphically represent in Fig. 5(c) the SCFG S_f obtained for f . The associated invariant is given in Fig. 5(b), along with the inferred guard.

5.1 Security Semantics

Our encoding of security semantics captures implicit flows (i.e., flows induced via the program control-flow structure) by constructing SCFGs that feature two *execution modes*, encoded with the help of a state variable \mathcal{Y} : (i) in *nominal* mode ($\mathcal{Y} = \text{ff}$), updates to security levels reflect explicit information flows, and (ii) in *upgrade analysis* mode ($\mathcal{Y} = \text{tt}$), the information flow from the *high* execution context pc to every variable updated in every possible execution path within the *Control Dependence Regions* (CDRs) of the current context are captured. A CDR ρ is a non-empty set of CFG (Control-Flow Graph) nodes that gathers every instruction that is control-dependent on a given branching statement. This use of CDRs is inspired by previous works [4, 5, 21, 27]⁴. The code in Fig. 5(a) features a single conditional branching statement on line 3, which induces the CDR ρ_1 . The junction of ρ_1 is the statement `output⊥(1)`. Two execution branches are possible within ρ_1 : one branch executes no statement, whereas the other performs the assignment `1 = 42` on line 4. This means that the execution of the latter is dependent on the condition on line 3. In the SCFG $S_{\mathcal{f}}$, the upgrade analysis of ρ_1 starts whenever the model reaches location ℓ_3 , which represents the *junction* of ρ_1 , if the branching statement that induces ρ_1 (encoded by ℓ_1) was subject to a high condition. We use a state variable hr to record the CDR currently subject to a high-condition. We make use of the input variable ω to abstract away the actual branch condition in nominal mode (since our security semantics abstracts away the values of program variables). This is for instance the case on location ℓ_1 in $S_{\mathcal{f}}$ when $\mathcal{Y} = \text{ff}$. The variable ω is also used to model upgrade analyses for multiple possible program paths which can be taken non-deterministically. In $S_{\mathcal{f}}$, this is the case on location ℓ_1 as well, when $\mathcal{Y} = \text{tt}$.

We give in Fig. 6 the set of translation rules that specify the security semantics of a program in terms of an SCFG. Each location of the resulting SCFG corresponds to a *semantic location*, that is defined as a pair (\mathbb{S}, ψ) where \mathbb{S} corresponds to a node in its CFG, and ψ is a *behavior mode* that belongs to $\{\text{njb}, \text{nb}\}$ (for nominal-or-junction and nominal behaviors, respectively). The junction step ψ is used in our encoding to distinguish the nominal mode from the upgrade analysis stage of junctions. Essentially, a semantic location that corresponds to a statement a that is the junction of a CDR behaves as a junction when $\psi = \text{njb}$, and according to a when $\psi = \text{nb}$. Thus, statements that are not junctions never give rise to semantic locations where $\psi = \text{nb}$. To clarify the translation rules, we define the helper predicate $\text{Junc}(\mathbb{S}, \psi)$ in Fig. 6 (where junc^{-1} is the retraction of junc : $\text{junc}^{-1}(\mathbb{S})$ gives the set of CDRs of which \mathbb{S} is the junction), that holds iff a semantic location (\mathbb{S}, ψ) represents an actual junction. We use $\text{target}(l)$ to denote the statement identified by a label l .

⁴ The classical algorithm of Ball [28] for computing CDRs works by identifying as a *junction* each dominating node in the post-dominator tree of the CFG. Such a junction j is reached by every execution path that starts from any node in the set ρ of nodes that j post-dominates. Further, one can always find a *unique* branching node that precedes nodes in ρ and belongs to every path from the source of the CFG to any node in ρ , and ρ is therefore a CDR.

$$\begin{array}{c}
\text{GOTO} \frac{\neg\text{Junc}(\ell) \quad \ell = (\langle\langle \text{goto } l \rangle\rangle; \mathbb{S}, _)}{\ell \xrightarrow{\text{tt}, \emptyset} (\text{target}(l), \text{njb})} \\
\text{SINK} \frac{\neg\text{Junc}(\ell) \quad \ell = (\langle\langle \text{output}_l(x) \rangle\rangle; \mathbb{S}, _) \quad \varphi(\ell) = (\bar{v} \text{ if } x = v, \bar{r} \text{ if } x = r) \sqsubseteq l \wedge pc \sqsubseteq l}{\ell \xrightarrow{\text{tt}, \emptyset} (\mathbb{S}, \text{njb})} \\
\text{ASSIGN} \frac{\neg\text{Junc}(\ell) \quad T_a}{\ell = (a; \mathbb{S}, _) \xrightarrow{\text{tt}, T_a} (\mathbb{S}, \text{njb})} \\
\text{BRANCH} \frac{\neg\text{Junc}(\ell) \quad \ell = (a; \mathbb{S}, _) \quad a = \langle\langle \text{if } (e) \text{ goto } l \rangle\rangle}{\begin{array}{l} \ell \xrightarrow{\omega \wedge \neg \Upsilon \wedge \bar{e} \sqsubseteq pc, \text{Brch}(\text{CDR}(a))} (\text{target}(l), \text{njb}) \quad \ell \xrightarrow{\omega \wedge \neg \Upsilon \wedge \bar{e} \sqsubseteq pc \vee \omega \wedge \Upsilon, \emptyset} (\text{target}(l), \text{njb}) \\ \ell \xrightarrow{\neg \omega \wedge \neg \Upsilon \wedge \bar{e} \sqsubseteq pc, \text{Brch}(\text{CDR}(a))} (\mathbb{S}, \text{njb}) \quad \ell \xrightarrow{\neg \omega \wedge \neg \Upsilon \wedge \bar{e} \sqsubseteq pc \vee \neg \omega \wedge \Upsilon, \emptyset} (\mathbb{S}, \text{njb}) \end{array}} \\
\text{JUNC} \frac{\text{Junc}(\ell) \quad \ell = (\mathbb{S}, \text{njb}) \quad J = \text{junc}^{-1}(\mathbb{S}) \quad P_J = \{P_\rho\}_{\rho \in J}}{\begin{array}{l} \ell \xrightarrow{hr \notin P_J, \emptyset} (\mathbb{S}, \text{nb}) \quad \ell \xrightarrow{\neg \Upsilon \wedge hr = P_\rho, \text{Start-ua}} (\text{inducing}(\rho), \text{njb})_{\rho \in J} \quad \ell \xrightarrow{\Upsilon \wedge hr \in P_J, \text{End-ua}} (\mathbb{S}, \text{nb}) \end{array}} \\
\text{CALL} \frac{\neg\text{Junc}(\ell) \quad T = \text{Effect}_m^{r,w} [pc \mapsto pc \sqcup \bar{r}]}{\ell = (\langle\langle r.m(w) \rangle\rangle; \mathbb{S}, \sigma) \xrightarrow{\text{tt}, T} (\mathbb{S}, \text{njb})} \quad \varphi_{\text{CALL}} \stackrel{\text{def}}{=} \text{Grd}_m^{r,w} [pc \mapsto pc \sqcup \bar{r}] \quad (\varphi\text{-CALL})
\end{array}$$

where:

$$\begin{array}{l}
\bar{p} \stackrel{\text{def}}{=} \perp \quad \bar{\ominus} e \stackrel{\text{def}}{=} \bar{e} \quad \bar{e} \oplus \bar{x} \stackrel{\text{def}}{=} \bar{e} \sqcup \bar{x} \quad \bar{r} \equiv \bar{s} \stackrel{\text{def}}{=} \bar{r} \sqcup \bar{s} \\
[l]_\Upsilon \stackrel{\text{def}}{=} (\text{if } \Upsilon \text{ then } \perp \text{ else } l) \sqcup pc \quad \bar{x} :=_\Upsilon l \stackrel{\text{def}}{=} \bar{x} := (\text{if } \Upsilon \text{ then } \bar{x} \text{ else } l) \sqcup pc \\
\text{Junc}(\mathbb{S}, \psi) \stackrel{\text{def}}{=} \text{junc}^{-1}(\mathbb{S}) \neq \emptyset \wedge \psi = \text{njb} \quad \text{Brch}(\rho) \stackrel{\text{def}}{=} [hr := P_\rho, pc := \top, \mathfrak{h}' := \mathfrak{h}] \\
\text{Start-ua} \stackrel{\text{def}}{=} [\Upsilon := \text{tt}, \mathfrak{h}' := \mathfrak{h}, \mathfrak{h} := \mathfrak{h}'] \quad \text{End-ua} \stackrel{\text{def}}{=} [hr := P_\perp, \Upsilon := \text{ff}, pc := \perp] \dot{\sqcup} \text{BulkUpgr}_{\mathfrak{h} \leftarrow \mathfrak{h}'}
\end{array}$$

Fig. 6: Translation rules and safety properties for encoding the security semantics.

The ASSIGN rule encodes the security semantics of assignments. We use \bar{e} to denote the security level of an expression e . from upgrade analyses in the rules. In nominal mode, $[l]_\Upsilon$ encodes the least upper-bound between l and the context level pc , and $\bar{x} :=_\Upsilon l$ models a *strong update* of the security level assigned to x with $[l]_\Upsilon$. In upgrade analysis mode, however, $[l]_\Upsilon$ is equal to the context level (i.e., \top), and $\bar{x} :=_\Upsilon l$ encodes a *weak update* of \bar{x} with pc . Then, a statement $v = r.f_p$ that loads a primitive field translates into a transition that updates \bar{v} with: the upper-bound between pc , \bar{r} , and the level of any object potentially pointed to by r as maintained by the heap abstraction \mathfrak{h} (i.e., \bar{r}) when in nominal mode; the upper-bound between pc and \bar{v} otherwise. BRANCH and JUNC encode the alternation of nominal and upgrade analyses, and do so with the help of a placeholder abstract heap \mathfrak{h}' that belongs to the same abstract heap domain as \mathfrak{h} , and is also represented with state variables. According to BRANCH, when a high branch is reached, the transformer $\text{Brch}(\rho)$: (i) sets the state variable hr used to record the CDR currently subject to a high-condition to P_ρ ; (ii) updates pc ; and (iii) stores the current heap abstraction to \mathfrak{h}' by copying the values of all variables $\mathbb{V}_{\mathcal{L}}$ (resp. $\mathbb{V}_{\mathcal{R}}$) to $\mathbb{V}_{\mathcal{L}}$ (resp. $\mathbb{V}_{\mathcal{R}}$). The join of abstract heaps that ends upgrade analyses is performed using a bulk upgrade. In effect, $\text{BulkUpgr}_{\mathfrak{h} \leftarrow \mathfrak{h}'}$: (i) upgrades the security typing environment for referenced portions of the heap according

Algorithm 1: SYNTHESIZEGUARD

Input: Method to analyze m
Result: Polymorphic information-flow guard Grd_m
 { Encode the security semantics of m as an SCFG S_m where every state variable related to the calling context is left uninitialized, and express the security requirement as a predicate $\varphi_m(\ell)$ on state variables for each location ℓ of S_m : }

- 1 $(S_m, \varphi_m) \leftarrow \text{ENCODE}(m)$
- 2 $\mathcal{B}_0 \leftarrow \{\ell \mapsto \neg\varphi_m(\ell) \mid \ell \in A(S_m)\}$ { Define all known unsafe states }
- 3 $\mathcal{B}_\infty \leftarrow \text{COREACH}(S_m, \mathcal{B}_0)$ { All states that are co-reachable to \mathcal{B}_0 }
- { Factor out the state variables that are not part of the calling context: }
- 4 $Grd_m \leftarrow \text{cofactor}(\neg\mathcal{B}_\infty(\ell_0(S_m)), X_0(S_m))$

to the result of the upgrade analysis in \mathfrak{h} , by joining every security level from \mathfrak{h} with the corresponding level in \mathfrak{h}' ; and (ii) restores every heap-related relation as saved in \mathfrak{h}' when entering the upgrade analysis mode. CALL encodes the security semantics of invocation of a method m based on its *polymorphic information-flow summary*, which is a *contract* that consists of:

- an information-flow *guard* Grd_m that specifies the invocation conditions under which the method call is secure, i.e., there is no illegal information flow in the method. This guard is described as constraints on the security types and heap structure of the method’s formal arguments;
- an *effect* $Effect_m$ about its *worst potential* side-effects on security levels and heap structure, that is in principle a *transformer* describing how the heap structure and security labels *may be* updated by the method.

We use the guard to enforce the desired security properties upon an invocation of m : this boils down to ensure that $\text{Inv.}(\varphi\text{-CALL})$ holds for the location ℓ in which m is called. The effect is used in CALL to update the typing environment and the heap model. In detail, $Grd_m^{r,w}$ and $Effect_m^{r,w}$ correspond to the aforementioned guard and effect—or a combination of several summaries in case of virtual method dispatch, where guards are combined using a conjunction, and transformers are merged—and after substitutions w.r.t. m ’s formal arguments. Furthermore, $e[v \mapsto l]$ denotes the substitution of security level expression l for variable v in e : this is required to upgrade the context pc w.r.t. the receiver object (the substitution in effects is performed in every expression on the right-hand side of assignments). For the sake of concision, we leave the computation of polymorphic effects out of the scope of this paper. In that respect, we want to mention that this computation is achievable, even for the cases of recursion, via an extension of our security semantics, accompanied by a dedicated processing of the co-reachability analysis results. Also note that a sound application of effects requires abstract heaps that capture object sharing relations, not just aliasing relations.

5.2 Guard Inference Procedure

We summarize the overall analysis procedure in Algorithm 1, where $\text{ENCODE}(m)$ denotes the specification of the security semantics of a method m as an SCFG

S_m and invariant φ_m as described above. We represent sets of states as mappings from locations to predicates on state variables, e.g., \mathcal{B}_0 is the set of all states that violate the invariant φ_m . \mathcal{B}_0 associates every location that corresponds to a sink statement with a predicate on security levels for program variables that violate security requirements encoded in φ_m . Then, the set of insecure states is back-propagated via a standard co-reachability analysis embodied by COREACH, i.e., finding all states from which a given set of states may be reached, and is typically solved using a fixed-point [23, 24]. On a symbolic finite-state system like S_f or S_m , this computation always terminates, and is traditionally performed using the least fixed-point (lfp)

$$\mathcal{B}_\infty \stackrel{\text{def}}{=} \text{lfp } \lambda \mathcal{B}_i. \mathcal{B}_0 \cup \text{pre}(\mathcal{B}_i), \quad (1)$$

where $\text{pre}(\mathcal{B})$ gives all predecessor states of \mathcal{B} . \mathcal{B}_∞ associates each location with a predicate that must *not* hold for every subsequent path in S_m to represent secure executions. Therefore, the guard for m can be obtained by complementing $\mathcal{B}_\infty(\ell_0)$ and eliminating every state variable that does not represent a proposition from m 's calling context. This is done with the help of $\text{cofactor}(f, g)$, which amounts to a partial evaluation of f w.r.t. all variables bound in g , i.e., this gives a predicate f' that does not involve any variable fully determined by g and $s.t$ ($g = \text{tt}$) \Rightarrow ($f = f'$).

Example 3 (Guard inference for m). The analysis first builds the SCFG given in Fig. 7, and associates the invariant $pc \sqcup \bar{b} \sqcup \vec{b} = \perp$ with location ℓ_3 . This states that, for m to be secure, this statement must be executed in a low context and given a low-sensitive reference \mathbf{b} (i.e., $\bar{b} = \perp$) that must only reach low-sensitive objects (i.e., $\vec{b} = \perp$). This gives the unsafe states shown in the first row of Table 3, where we report a trace of the co-reachability analysis and the resulting guard for each domain. The guard obtained with the **dumb** domain is the least precise of all three, as it basically describes m as insecure if it is called in high context, or whenever any of its effective arguments or objects they may reach in the heap is high-sensitive. With this domain, the statement $\mathbf{r}.\mathbf{fa} = \mathbf{a}$ (location ℓ_2) may raise the security level \vec{b} since $\mathbf{r} \sim \mathbf{b} \in \mathbb{V}_{\text{tt}}$ (as $\text{CanRelate}(\mathbf{r} \sim \mathbf{b}) = \text{maybe}$).

Table 3: Polymorphic guard inference for method m , for different heap domains.

	with $\mathcal{B}_0 = \{\ell_3 \mapsto pc \sqcup \bar{b} \sqcup \vec{b} \neq \perp\} \cup \{\ell_i \mapsto \text{ff}\}_{i \in \{0,1,2,4\}}$
deep	$\mathcal{B}_1 = \mathcal{B}_0 \cup \{\ell_2 \mapsto pc \sqcup \bar{b} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \sim \mathbf{r} \text{ then } \bar{\mathbf{a}} \sqcup \bar{\mathbf{a}} \text{ else } \perp)\} \neq \perp$
	$\mathcal{B}_2 = \mathcal{B}_1 \cup \{\ell_1 \mapsto pc \sqcup \bar{b} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \sim \mathbf{r} \text{ then } \bar{\mathbf{a}} \sqcup \bar{\mathbf{a}} \sqcup \bar{\mathbf{i}} \text{ else } \perp) \sqcup (\text{if } \mathbf{b} \xrightarrow{*} \mathbf{a} \text{ then } \bar{\mathbf{i}} \text{ else } \perp)\} \neq \perp$
	$\mathcal{B}_\infty = \mathcal{B}_2 \cup \{\ell_0 \mapsto pc \sqcup \bar{b} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \xrightarrow{*} \mathbf{a} \text{ then } \bar{\mathbf{i}} \text{ else } \perp)\} \neq \perp$
	$\text{Grd}_m = pc \sqcup \bar{b} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \xrightarrow{*} \mathbf{a} \text{ then } \bar{\mathbf{i}} \text{ else } \perp) = \perp$
shal	$\mathcal{B}_1 = \mathcal{B}_0 \cup \{\ell_2 \mapsto pc \sqcup \bar{b} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \sim \mathbf{r} \text{ then } \bar{\mathbf{a}} \sqcup \bar{\mathbf{a}} \text{ else } \perp)\} \neq \perp$
	$\mathcal{B}_2 = \mathcal{B}_1 \cup \{\ell_1 \mapsto pc \sqcup \bar{b} \sqcup \bar{\mathbf{i}} \sqcup \vec{b} \sqcup (\text{if } \mathbf{b} \sim \mathbf{r} \text{ then } \bar{\mathbf{a}} \sqcup \bar{\mathbf{a}} \sqcup \bar{\mathbf{i}} \text{ else } \perp)\} \neq \perp$
	$\mathcal{B}_\infty = \mathcal{B}_2 \cup \{\ell_0 \mapsto pc \sqcup \bar{b} \sqcup \bar{\mathbf{i}} \sqcup \vec{b}\} \neq \perp$
	$\text{Grd}_m = pc \sqcup \bar{b} \sqcup \bar{\mathbf{i}} \sqcup \vec{b} = \perp$
dumb	$\mathcal{B}_1 = \mathcal{B}_0 \cup \{\ell_2 \mapsto pc \sqcup \bar{b} \sqcup \bar{\mathbf{a}} \sqcup \vec{b} \sqcup \bar{\mathbf{a}} \neq \perp\}$
	$\mathcal{B}_2 = \mathcal{B}_1 \cup \{\ell_1 \mapsto pc \sqcup \bar{b} \sqcup \bar{\mathbf{a}} \sqcup \bar{\mathbf{i}} \sqcup \vec{b} \sqcup \bar{\mathbf{a}} \neq \perp\}$
	$\mathcal{B}_\infty = \mathcal{B}_2 \cup \{\ell_0 \mapsto pc \sqcup \bar{b} \sqcup \bar{\mathbf{a}} \sqcup \bar{\mathbf{i}} \sqcup \vec{b} \sqcup \bar{\mathbf{a}} \neq \perp\}$
	$\text{Grd}_m = pc \sqcup \bar{b} \sqcup \bar{\mathbf{a}} \sqcup \bar{\mathbf{i}} \sqcup \vec{b} \sqcup \bar{\mathbf{a}} = \perp$

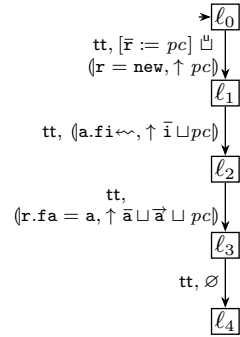


Fig. 7: SCFG for m .

$$\begin{array}{c}
\text{M-UPGRADE} \frac{\ell \xrightarrow{g, T} \ell'}{(\ell, \mathcal{V}, \bar{h}) \xrightarrow{\llbracket \mathcal{I} \wedge g \rrbracket T} (\ell', \mathcal{V}, \bar{h})}, \text{M-JUNC} \frac{\text{Junc}(\ell) \quad \ell \xrightarrow{g, T} \ell'}{(\ell, \mathcal{V}, \bar{h}) \xrightarrow{\llbracket \neg \mathcal{I} \wedge g \rrbracket T} (\ell', \mathcal{V}, \bar{h})} \\
\text{M-BRCH} \frac{\neg\text{Junc}(\ell) \quad \ell = (\langle\langle \text{if}(e) \text{ goto } _ \rangle\rangle; _ , _) \quad c = \mathcal{V}(e)}{\ell \xrightarrow{\tau \wedge \neg \mathcal{I} \wedge g_{tt}, T_{tt}} \ell_{tt} \quad \ell \xrightarrow{\neg \tau \wedge \neg \mathcal{I} \wedge g_{ff}, T_{ff}} \ell_{ff}}{(\ell, \mathcal{V}, \bar{h}) \xrightarrow{\llbracket \neg \mathcal{I} \wedge g_c \rrbracket T_c} (\ell_c, \mathcal{V}, \bar{h})}, \\
\text{M-STM} \frac{\neg\text{Junc}(\ell) \quad \ell = (a; _ , _) \xrightarrow{g, T} \ell' \quad a \in \text{Stm} \\ \mathcal{V}', \bar{h}' = \text{Apply}_a(\mathcal{V}, \bar{h})}{(\ell, \mathcal{V}, \bar{h}) \xrightarrow{\llbracket \neg \mathcal{I} \wedge g \rrbracket T} (\ell', \mathcal{V}', \bar{h}')}
\end{array}$$

where

$$\text{Stm} \stackrel{\text{def}}{=} \left\{ \langle\langle v = e \rangle\rangle, \langle\langle v = r.f_p \rangle\rangle, \langle\langle r = s \rangle\rangle, \langle\langle r = s.f_r \rangle\rangle, \langle\langle r.f_p = e \rangle\rangle, \langle\langle r.f_r = s \rangle\rangle, \right. \\
\left. \langle\langle r = \text{new } _ \rangle\rangle, \langle\langle r = \text{null} \rangle\rangle, \langle\langle \text{goto } _ \rangle\rangle, \langle\langle \text{output}_i(_) \rangle\rangle \right\}$$

and

$$\text{Apply}_a(\mathcal{V}, \bar{h}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{V}[v \mapsto \mathcal{V}(e)], \bar{h} & \text{if } a = \langle\langle v = e \rangle\rangle \\ \mathcal{V}[v \mapsto r.f_p], \bar{h} & \text{if } a = \langle\langle v = r.f_p \rangle\rangle \\ \mathcal{V}, \bar{h} & \text{if } a \in \{ \langle\langle \text{goto } _ \rangle\rangle, \langle\langle \text{output } _ (_) \rangle\rangle \} \\ \mathcal{V}, [a]_{\bar{h}} & \text{otherwise.} \end{cases}$$

Fig. 8: Full semantics.

On the other hand, the inference with `shal` is able to distinguish whether `b` and `r` may alias on location ℓ_2 , and then rules this case out thanks to the statement `r = new B` (location ℓ_0). However, the guard does not hold whenever $\bar{i} \neq \perp$, as the domain cannot distinguish whether `b.fa` aliases `a` or not: therefore, the statement `a.fi = i` (location ℓ_1) always raises the level \bar{b} to that of $pc \sqcup \bar{i}$. At last, the domain `deep` distinguishes whether `b.fa` aliases `a` or not, and the guard indicates that `m` may not be secure if `i` is high-sensitive and `a` and `b` relate to each other via `b.fa`.

6 Soundness

We prove that any program guarded with a security guard inferred by our method guarantees termination-insensitive noninterference [3]. This notion states that, for any initial states q and q' whose secret parts may only differ, the observation sequences of the program running from the states q and q' will either be the same, or one is a prefix of the other. The reason for the latter case is that this notion is a termination-insensitive property.

To prove noninterference, we first define the full semantics of a program by an SCFG that extends the security semantics with its operational semantics. The full semantics basically extends the security semantics of a program with

its execution semantics. In addition to the symbolic variables that belong to the security semantics, we consider a concrete model for the *heap*, as well as local variables. A *configuration* is therefore defined as $(\ell, \mathcal{V}, \tilde{h})$ where ℓ denotes the semantic location, \mathcal{V} is a *valuation* for all primitive variables, \tilde{h} is the heap value of the concrete heap domain \mathbb{H}_{ct} . We give the full semantics in Fig. 8,

where $\xrightarrow{g, T}$ and $\xrightarrow{\boxed{g} \boxed{T}}$ are transitions of the security semantics and of the full semantics, respectively. Apart from M-UPGRADE and M-SINK, every rule in this figure applies only when the SCFG is in the nominal mode (i.e., $T = \text{ff}$) — this is reflected in the guards for the full semantics. According to rule M-UPGRADE, only the updates by the security semantics are included when the program runs in the upgrade analysis mode (i.e., $T = \text{tt}$), i.e., the program statements run in nominal mode only. The semantics of a branch statement is defined based on its corresponding rule of the security semantics with the addition that the input variable τ is substituted by the concrete guard e of the statement. The rule M-STM handles assignments, and other non-branching control-flow related statements, by extending the updates by the security semantics with the updates to the primitives \mathcal{V} and the concrete heap \tilde{h} .

Let \mathcal{S} denote the (symbolic) full semantics of a program, and $\llbracket \mathcal{S} \rrbracket = \langle \mathcal{Q}, \mathcal{I}, \rightarrow, \mathcal{Q}_0 \rangle$ be an automaton that describes its concrete semantics, where \mathcal{Q} is the set of states, \mathcal{I} is the set of inputs, $\rightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ is the set of transitions and \mathcal{Q}_0 is the set of initial states. A program state q is defined as $\langle \ell, \mathcal{V}, \Omega, \mathcal{X}, \mathbf{h} \rangle$ where ℓ is the current location, \mathcal{V} is the valuation for every one of the primitive program variables P , $\Omega: P \cup R \rightarrow \mathbb{L}$ is the *security typing environment* for the primitive variables P and references R , and \mathcal{X} is the current valuation of the state variables of the security semantics except the location and the typing environment for the references. We use the notation $\mathcal{X}_{\mathfrak{h}}$ to show \mathcal{X} 's heap abstraction \mathfrak{h} , and $\mathcal{X}_{\vec{\tau}}$ to denote its security typing environment. Furthermore, \mathbf{h} is the valuation of the concrete heap's variables. Let $q \xrightarrow{\eta}_* q'$ be an execution of full semantics with a non-zero length (i.e., the reflexive and transitive closure of the concrete transition relation \rightarrow) from the state q to the state q' , where $\eta \in \{o, \perp\}$. This execution either ends by executing a statement that outputs on a channel (i.e., $\eta = o$) or makes no observation (i.e., $\eta = \perp$). We denote an execution that never reaches an observation point by $q \xrightarrow{\perp}_*$. We define noninterference based on a *low-equivalence relation*, that states that the public parts of the two states q_1 and q_2 are indistinguishable.

Definition 4 (Indistinguishable Stores). *We say two valuations $\mathcal{V}_1 \in \text{Val}(P)$ and $\mathcal{V}_2 \in \text{Val}(P)$ are low-equivalent w.r.t. the typing environment $\Omega: P \cup R \rightarrow \mathbb{L}$, denoted by $\mathcal{V}_1 =_{\Omega} \mathcal{V}_2$, iff $\mathcal{V}_1(v) = \mathcal{V}_2(v)$ for all $v \in P$ where $\Omega(v) = \perp$.*

Definition 5 (Low-Bisimulation). *We say two states $q_i = \langle \ell, \mathcal{V}_i, \Omega_i, \mathcal{X}_i, \mathbf{h}_i \rangle$, $i \in \{1, 2\}$ are compatible, denoted by $q_1 \approx q_2$, iff (i) $\Omega_1 = \Omega_2$, (ii) $\mathcal{X}_{1, \vec{\tau}} = \mathcal{X}_{2, \vec{\tau}}$, (iii) $\mathcal{V}_1 =_{\Omega_1} \mathcal{V}_2$, and (iv) $\mathbf{h}_1 =_{\mathcal{X}_{1, \mathfrak{h}}} \mathbf{h}_2$. They are called low-bisimilar, denoted by $q_1 \sim_{\text{low}} q_2$, iff $q_1 \approx q_2$, and if $q_1 \xrightarrow{o}_* q'_1$, then either (a) there exists q'_2 such that $q_2 \xrightarrow{o}_* q'_2$ and $q'_1 \sim_{\text{low}} q'_2$, or (b) $q_2 \xrightarrow{\perp}_*$, and vice versa.*

Theorem 2 (Noninterference). *For any method m guarded by a security guard Grd_m , and any initial states q_1 and q_2 where $q_1 \approx q_2$ and $q_i \models Grd_m$, $i \in \{1, 2\}$, it holds $q_1 \sim_{low} q_2$.*

Proof. To prove this theorem, we show that there exists a witnessing bisimulation relation for $q_1 \sim_{low} q_2$. See Appendix B.

7 Implementation and Evaluation

To empirically validate our approach, we assess the respective performances of our three heap domains on actual code, both in terms of precision and scalability.

7.1 Implementation Details

We have first implemented a tool that relies on `soot` [16] to obtain the `Jimple` code of a program, and translates it into our input language. `Jimple` is an intermediate language to represent `Java` byte-code at a higher level. The semantics of its instructions and reference manipulations correspond to that of the `JVM`. One `Jimple` statement roughly translates into one statement of our input language. We have then implemented the guard inference algorithm in a prototype tool called `Guardies`⁵, that features multiple instantiations of our heap domains. `Guardies`'s pre-analysis relies on a naive analysis of the class hierarchy to construct a graph that allows us to compute facts about heap-related relations (i.e., function `CanRelate`). This tool relies on `ReaX` [30] to solve the co-reachability problems. `ReaX` uses (Multi-terminal) Binary Decision Diagrams—(MT)BDDs— [31, 32] to represent symbolic expressions and compute the underlying fixed-points. To deal with guards and transformers that encode the semantics of library methods, we rely on `stubs`, given to `Guardies`, that describe the effects of these methods at a high level. We manually defined the security semantics of methods from the standard `Java` and `Android` libraries (about 1200 methods in total) in this way.

7.2 Precision & Recall

We have employed the `IFSPEC` benchmark suite [25] to assess the *precision* of our different heap domains and compare our results to `KEY` [33], `CASSANDRA` [21], and `JOANA` [17]. The *precision* refers to a proportion of test cases that are correctly classified. The *recall* is the fraction of true positive and false negative test cases that are categorized correctly. `IFSPEC` provides 232 test cases that showcase various information-flow vulnerabilities in `Java` programs, with various forms of explicit and implicit information leaks. We report in Table 4 the precision results that we obtain for different abstract heap domains for the various categories of leaks and language features that `IFSPEC` covers. We have checked 164 out of 232 test cases supported by our sub-language: the excluded cases involve reflection,

⁵ Available as a software artifact [29], with user documentation and source code at <http://nberth.space/symmaries>.

static class initializers, exceptions and method calls (11, 10, 9, and 39 samples respectively—we have excluded all cases in the latter category as they check the ability of the analysis to handle information-flows across method calls, while we left the problem of computing method effects aside). Note that a test case may belong to multiple categories.

Since our approach is sound, we obtain 100% *recall*, i.e., we correctly detect every insecure flow. Regarding precision, our experiments show that all the domains have close precision: the

Table 4: IFSPEC Precision Results

Category	#Smpls	deep	shal	dumb	KEY	JOANA	CASSANDRA
explicit-flows	143	80.4	79.7	78.3	70.6	77.6	72.7
implicit-flows	21	71.4	71.4	71.4	57.1	57.1	61.7
simple	51	72.5	72.5	72.5	64.7	76.4	68.6
high-cond.	10	80	80	80	60	60	60
arrays	26	73	73	69.2	65.3	76.9	69.2
library	69	88.4	88.4	86.9	76.8	76.7	79.7
aliasing	7	71.4	71.4	57.1	57.1	42.8	42.8
average		79.2	78.6	77.4	68.9	75	71

The #Smpls column shows the number of included samples for each category; other figures are percentages.

deep domain offers the highest precision of 79.2% and **dumb** offers the lowest precision of 77.4%. The false positives (i.e., the secure test cases that were restrictively classified as insecure) mainly occur because our domains are field-insensitive, value-insensitive, do not distinguish elements in some collections of data, or due to the over-approximations in heap-related relations. The results for the aliasing category are rather similar; 3 test cases in this category are insecure that are classified correctly by all three domains, as our analysis is sound. Two of the remaining 4 secure test cases are classified as insecure in all domains due to value- and field-insensitivity.

On average, the domain **deep** offers the best precision in five categories. It slightly underperforms the state-of-the-art for simple and arrays test cases only, notably due to value- and field-insensitivity. In some categories, the improvement is noticeable, i.e., it improves the best precision of the aliasing category offered by the existing tools by 14.3%, improves the library category by 8.7% and enhances the implicit-flows category by 9.5%. We attribute these substantial results in part to our precise handling of implicit flows across method calls (unlike CASSANDRA which forbids method calls in high-contexts for instance), and in part to our heap abstract domain, that is able to precisely track some intricate aliasing relations. That most of our domains obtain similar precision results on the aliasing category may indicate that these test cases are rather uniform in the facts about aliasing that need to be discovered to detect secure cases. Our findings show that while different domains had close precision results, they offer different computational complexity though. Further, IFSpec only partially covers the set of IFC problems one can encounter in practice; we therefore refrain from generalizing our results. Yet, IFSpec is the most extensive benchmark available for IFC that we know of.

7.3 Scalability Evaluation

We have conducted experiments on real-life web applications to compare different heap abstract domains in terms of scalability. To accommodate computationally intensive analyses, we interrupt any analysis after 5 minutes or if it uses more than 4GB of memory. We have used applications from the ABM benchmark [26], a

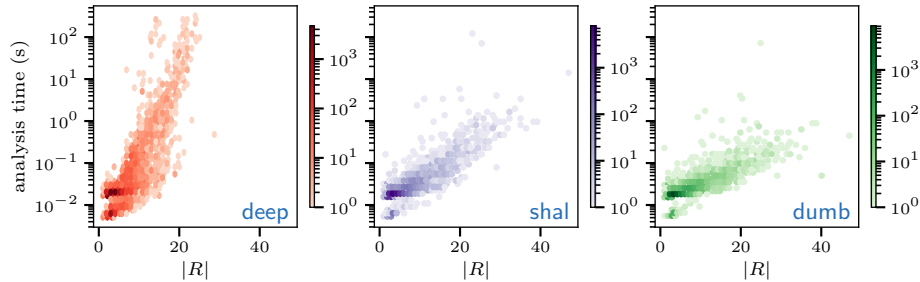


Fig. 9: Density plots showing the distributions of analyzed ABM methods w.r.t. both the number of reference variables (horizontal axes) and the analysis time (vertical axes). Note the shared log scales, including on the color-bars.

collection of 139 open-source projects that is dedicated to the evaluation of static analyzers for **Java** applications. Its content is deemed representative of real-world software, and has already been used for evaluating static taint analysis and dead code elimination approaches [26]. From this collection, we extracted the **Java** code from 60 applications with sizes ranging from 133 to 25K lines of **Java**. This provided us with a total of 22,512 analyzable methods. Overall, the **deep** domain led to 146 analyses being interrupted due to the timeouts or memory limitations (3 for **shal**, 0 for **dumb**). We plot in Fig. 9, for each domain, the distributions of successful analyses w.r.t. the number of reference variables and analysis time. As expected, analysis times grow with the amount of references, and by factors that depend on the heap-related relations captured flow-sensitively by the domains, e.g., **deep** is more expensive compared to **dumb** and **shal**. Further, many methods have fewer than 10 reference variables, and as a result most analysis times do not exceed 0.1s for every domain. Those figures empirically support the applicability of our approach on real-life applications.

Note that an ideal study on the scalability of different heap domains would compare the domains under different analysis techniques, provided by different tools. This, however, requires the support of the existing tools for modeling different heap domains. To the best of our knowledge, there was no such tool, as each implementation is typically tied with its own heap model, if any at all. Otherwise, extending the tools to support different heap domains is virtually infeasible since most existing tools rely on store-based models. To further compare the respective precision of each domain, we also report in Fig. 10 the ratios of unsatisfiable and tautological guards obtained for each domain. We observe that the precision of

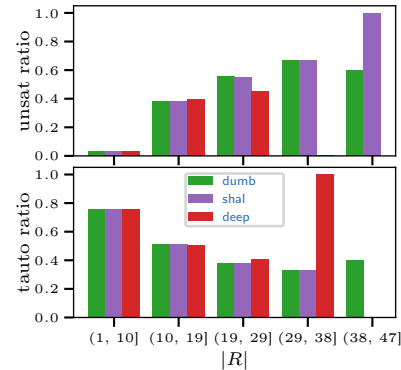


Fig. 10: Plots showing for each domain, the proportions of unsatisfiable (top) and tautological (bottom) guards vs number of reference variables.

all domains seems similar when the number of reference variables is low, and diverges with growing numbers of references. Moreover, `deep` appears to be more permissive than `dumb` and `shal` for methods with many references. However, that `deep` did not produce unsatisfiable guards for methods with more than 29 references indicates that many analyses of such methods were interrupted.

8 Related Works

Static analysis approaches to ensure noninterference have been studied extensively in the community. The vast majority of suggested IFC solutions concentrates on type-systems [2–4], and various tools that target realistic programming languages have been developed for verifying such properties. Prominent examples include JFlow JIF [3], FlowCaml [2], CASSANDRA [21], and KEY [33]. Albeit sound, the aforementioned approaches often lack precision in practice (e.g., [21]), or require user intervention, such as the specification of loop invariants (e.g., [3]). Another line of research trades efficiency for soundness and/or precision, by exploiting more generic techniques like interprocedural dataflow analysis [34] or program slicing [35]. JOANA [17], DroidSafe [18] and FLOWDROID [19] are prominent frameworks in this category. Other solutions are dedicated to web applications [36] or Android apps [37], although most of them do not handle implicit flows or lack soundness (e.g., [17–19]). Tools that provide sound results via other forms of global program analyses include HORNDROID [38, 39], which does not capture implicit flows. In contrast to the above methods, our approach is proven sound, captures implicit flows (via heap), and our experiments show that it improves the state-of-the-art precision. Further, the above approaches often rely on a simple store-based representation of the heap specified as a mapping from references (or abstract locations) to heap locations [4, 21, 40], or do not rely on a flow-sensitive heap abstraction [21]. By contrast, we use a store-less representation, where the structure of the heap is specified using a parameterizable family of (possibly over-approximated) relations. *This offers different levels of over-approximation and complexity, enabling the user to easily trade-off performance and scalability.*

Few works have addressed the problem of capturing implicit flows while exploiting flow-sensitive heap abstractions [14, 41, 42]. Khakpour [41] synthesizes sound security monitors that enforce IFC by using a symbolic discrete control algorithm. This work operates intraprocedurally on high-level programs and uses an *ad hoc* field-sensitive heap abstraction that does not scale well. Zanioli et al. [14] advance an abstract-interpretation-based analysis, where the construction of the heap abstraction is delegated to a separate analysis. Their analysis can operate on a flow-sensitive abstraction as produced by a TVLA-based shape analysis [43], yet it can only be applied to small, high-level programs. Other forms of symbolic heap abstractions have already been used in static program analysis. Separation logic [44] models a heap as a formula that comprises atomic predicates combined using the *separation* operator. While we use a store-less representation of the heap expressed using a proposition, symbolic heaps in separation logic are store-based, more expressive, and consequently are more complex for verification.

Store-less heap abstractions are also polymorphic and enable us to operate on each method of the program in isolation. This is to be contrasted with traditional data-flow analysis [34], where flow functions must be distributive and expressed on finite domains (as typically provided by store-based abstractions).

9 Conclusion

We have introduced a generic abstract heap domain for modeling heaps and information flow via heap for low-level object-oriented programs, and instantiated it with different families of relations. Our experiments showed that our instantiated heap models improve the state-of-the-art precision, and that the precision has an inverse relationship with scalability. We are currently investigating the computation of method summaries in order to obtain a fully modular interprocedural IFC analysis. **Guardies** can be improved by implementing a more advanced analysis to reduce the amount of symbolic variables involved to represent the heap, thereby improving scalability. Further, the instantiated heap domains are field-insensitive, and a natural extension is introducing support for field-sensitive analyses.

Acknowledgements The first author was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/M027287/1, and the second author was supported by the Swedish Knowledge Foundation (KKs) via the grant 20160186.

References

1. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20, Los Alamitos, CA, USA, April 1982. IEEE Computer Society. <https://doi.org/10.1109/SP.1982.10014>.
2. François Pottier and Vincent Simonet. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003. ISSN 0164-0925. <https://doi.org/10.1145/596980.596983>. URL <http://doi.acm.org/10.1145/596980.596983>.
3. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. ISSN 0733-8716. <https://doi.org/10.1109/JSAC.2002.806121>.
4. Gilles Barthe, David Pichardie, and Tamara Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *Proceedings of the 16th European Symposium on Programming, ESOP’07*, pages 125–140, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71314-2. URL <http://dl.acm.org/citation.cfm?id=1762174.1762189>.
5. Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 2010 14th European Conference on Software Maintenance*

- and *Reengineering*, CSMR '10, pages 146–155, USA, 2010. IEEE Computer Society. ISBN 9780769543215. <https://doi.org/10.1109/CSMR.2010.26>. URL <https://doi.org/10.1109/CSMR.2010.26>.
6. Daniel Hedin and Andrei Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4718-3. <https://doi.org/10.1109/CSF.2012.19>. URL <http://dx.doi.org/10.1109/CSF.2012.19>.
 7. Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.
 8. Tachio Terauchi and Alex Aiken. Secure Information Flow As a Safety Problem. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 352–367, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28584-9, 978-3-540-28584-7. https://doi.org/10.1007/11547662_24. URL http://dx.doi.org/10.1007/11547662_24.
 9. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proceedings of the 17th International Conference on Formal Methods*, FM'11, pages 200–214, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642214363.
 10. Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing*, pages 193–209, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32004-3.
 11. Masaaki Mizuno and David Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Form. Asp. Comput.*, 4(1):727–754, November 1992. <https://doi.org/10.1007/BF03180570>. URL <https://doi.org/10.1007/BF03180570>.
 12. Mirko Zanotti. Security typings by abstract interpretation. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 360–375, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540442359.
 13. Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. <https://doi.org/10.1145/964001.964017>. URL <https://doi.org/10.1145/964001.964017>.
 14. Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: Static analysis of information leakage with sample. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1308–1313, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450308571. <https://doi.org/10.1145/2245276.2231983>. URL <https://doi.org/10.1145/2245276.2231983>.

15. Vini Kanvar and Uday P. Khedker. Heap Abstractions for Static Analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, June 2016. ISSN 0360-0300. <https://doi.org/10.1145/2931098>. URL <http://doi.acm.org/10.1145/2931098>.
16. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.
17. Christian Hammer and Gregor Snelting. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *Int. J. Inf. Secur.*, 8(6):399–422, October 2009. ISSN 1615-5262. <https://doi.org/10.1007/s10207-009-0086-1>. URL <http://dx.doi.org/10.1007/s10207-009-0086-1>.
18. Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL <https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe>.
19. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. <https://doi.org/10.1145/2594291.2594299>. URL <http://doi.acm.org/10.1145/2594291.2594299>.
20. Andrew Johnson, Lucas Wayne, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, pages 291–302, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. <https://doi.org/10.1145/2737924.2737957>. URL <https://doi.org/10.1145/2737924.2737957>.
21. Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM '14*, pages 93–104, New York, NY, USA, 2014. ACM. ISBN 9781450331555. <https://doi.org/10.1145/2666620.2666631>. URL <https://doi.org/10.1145/2666620.2666631>.
22. Gérard Boudol. Secure information flow as a safety property. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 20–34, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-01465-9.

23. Peter J. G. Ramadge and W. Murray Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989. <https://doi.org/10.1109/5.21072>.
24. Amir Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. <https://doi.org/10.1145/75277.75293>. URL <https://doi.org/10.1145/75277.75293>.
25. Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. A uniform information-flow security benchmark suite for source code and bytecode. In Nils Gruschka, editor, *Secure IT Systems*, pages 437–453, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03638-6.
26. Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. Toward an Automated Benchmark Management System. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 13–17, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343855. <https://doi.org/10.1145/2931021.2931023>. URL <https://doi.org/10.1145/2931021.2931023>.
27. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. <https://doi.org/10.1145/359636.359712>. URL <https://doi.org/10.1145/359636.359712>.
28. Thomas Ball. What's in a Region? Or Computing Control Dependence Regions in near-Linear Time for Reducible Control Flow. *ACM Lett. Program. Lang. Syst.*, 2(1–4):1–16, March 1993. ISSN 1057-4514. <https://doi.org/10.1145/176454.176456>. URL <https://doi.org/10.1145/176454.176456>.
29. Nicolas Berthier and Narges Khakpour. Artifact for Paper (Symbolic Abstract Heaps for Polymorphic Information-flow Guard Inference), September 2022. URL <https://doi.org/10.5281/zenodo.7103855>.
30. Nicolas Berthier and Hervé Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *12th Int. Workshop on Discrete Event Systems*, WODES '14, pages 46–53. IFAC, May 2014. ISBN 978-3-902823-61-8. <https://doi.org/10.3182/20140514-3-FR-4046.00099>.
31. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986. ISSN 0018-9340. <https://doi.org/10.1109/TC.1986.1676819>. URL <https://doi.org/10.1109/TC.1986.1676819>.
32. J. P. Billon. Perfect normal forms for discrete programs. Technical report, Bull, 1987.
33. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Softw. Syst. Model.*, 4

- (1):32–54, February 2005. ISSN 1619-1366. <https://doi.org/10.1007/s10270-004-0058-x>. URL <https://doi.org/10.1007/s10270-004-0058-x>.
34. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95*, pages 49–61, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. <https://doi.org/10.1145/199448.199462>. URL <http://doi.acm.org/10.1145/199448.199462>.
 35. John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7(3):305–317, September 1977. ISSN 0001-5903. <https://doi.org/10.1007/BF00290339>. URL <http://dx.doi.org/10.1007/BF00290339>.
 36. Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671. ACM, 2014.
 37. Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.
 38. S. Calzavara, I. Grishchenko, and M. Maffei. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *2016 IEEE European Symposium on Security and Privacy (EuroSecP)*, pages 47–62, March 2016. <https://doi.org/10.1109/EuroSP.2016.16>.
 39. S. Calzavara, I. Grishchenko, A. Koutsos, and M. Maffei. A Sound Flow-Sensitive Heap Abstraction for the Static Analysis of Android Applications. In *30th Computer Security Foundations Symposium, CSF'17*, pages 22–36. IEEE, August 2017. <https://doi.org/10.1109/CSF.2017.19>.
 40. Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 91–102. ACM, 2006. <https://doi.org/10.1145/1111037.1111046>. URL <https://doi.org/10.1145/1111037.1111046>.
 41. Narges Khakpour. A field-sensitive security monitor for object-oriented programs. *Comput. Secur.*, 108:102349, 2021. <https://doi.org/10.1016/j.cose.2021.102349>. URL <https://doi.org/10.1016/j.cose.2021.102349>.
 42. Narges Khakpour and Charilaos Skandylas. Synthesis of a Permissive Security Monitor. In *European Symposium on Research in Computer Security*, pages 48–65. Springer, 2018.
 43. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, May 2002. ISSN 0164-0925. <https://doi.org/10.1145/514188.514190>. URL <https://doi.org/10.1145/514188.514190>.

44. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL <http://dl.acm.org/citation.cfm?id=645683.664578>.

A Proof of Secure Heap Abstraction

A.1 Sound Security Typing

We first define the soundness of security typing for an abstract heap \mathfrak{h} from a domain defined using aliasing and field-aliasing relations as follows:

Definition 6 (Sound Security Typing). An abstract heap $\mathfrak{h} \in \mathbb{H}_d$ for $d \in \{\text{deep}, \text{shal}, \text{dumb}\}$, is correctly typed iff:

- any pair of aliasing references have identical security levels:
 $\varphi_{\sim} \stackrel{\text{def}}{=} \forall (r, s) \in R^2, \mathfrak{h} \models r \sim^{\mathfrak{h}} s \Rightarrow \mathfrak{h} \models \vec{r} = \vec{s}$; and
- the security level associated with a reference may only be greater or equal than that of the references its fields may (transitively) alias:

$$\varphi_{\dot{\rightarrow}^* \sqsupseteq} \stackrel{\text{def}}{=} \forall (r, s) \in R^2, \mathfrak{h} \models r \dot{\rightarrow}^* s \Rightarrow \mathfrak{h} \models \vec{r} \sqsupseteq \vec{s}.$$

A predicate is an inductive invariant for a heap transformer of $\text{HeapDom}_{\text{hd}}$, if it is preserved by the transformer:

Definition 7 (Inductive Invariant). A predicate φ is an inductive invariant for a transformer $T \in \mathbb{T}_{\text{hd}}$ iff $\forall \mathfrak{h} \in \mathbb{H}_{\text{hd}}, (\mathfrak{h} \models \varphi \Rightarrow \llbracket T \rrbracket \mathfrak{h} \models \varphi)$.

We say a predicate is an inductive invariant for a heap domain $\text{HeapDom}_{\text{hd}}$, if it is an inductive invariant for all its transformers.

Proposition 1. The predicates φ_{\sim} and $\varphi_{\dot{\rightarrow}^* \sqsupseteq}$ are inductive invariants for the abstract heap domain $\text{HeapDom}_{\text{deep}}$.

A.2 Proof of Theorem 1

Proof. Let $\mathfrak{h}_1, \mathfrak{h}_2 \sqsubseteq \mathbb{H}_{\text{crt}}, \mathfrak{h} \in \mathbb{H}_{\text{deep}}, \mathfrak{h}_1 =_{\mathfrak{h}} \mathfrak{h}_2$ where \mathbb{H}_{crt} is the set of concrete heaps and \mathbb{H}_{deep} is the set of abstract heaps from the domain `deep`. According to Def. 3, we should prove that for any operation (as, as') of reference assignment statements and their corresponding operations on abstract heaps where $as \in \{\langle\langle r = s \rangle\rangle, \langle\langle r = s.f_r \rangle\rangle, \langle\langle r = \text{null} \rangle\rangle\}$, $\mathfrak{h}'_1 =_{\mathfrak{h}'} \mathfrak{h}'_2$ holds where $\mathfrak{h}'_i = \llbracket as \rrbracket_{\mathfrak{h}_i}$, $i \in \{1, 2\}$, and $\mathfrak{h}' = \llbracket as' \rrbracket$.

Let $G_R^{\mathfrak{h}_i} \stackrel{\text{def}}{=} (N_{\mathfrak{h}_i}, E_{\mathfrak{h}_i}), i \in \{1, 2\}$. From $\mathfrak{h}_1 =_{\mathfrak{h}} \mathfrak{h}_2$, it follows that $G_{\perp}^{\mathfrak{h}_1} \cong G_{\perp}^{\mathfrak{h}_2}$ from Definition 2. We prove that $G_{\perp}^{\mathfrak{h}'_1} \cong G_{\perp}^{\mathfrak{h}'_2}$ by a case analysis on as . Since, the proof of all cases are very similar and follows the same strategy, we limit ourselves to proving the case of reference assignment. For the proof of $r = s$, two cases can happen:

- (i) If $\vec{s} = \perp$, this means that $s \in N_{\tilde{h}_i}, i \in \{1, 2\}$. The truth value of the variables $r' \sim s$ and $s \xrightarrow{f} r'$ respectively determine the presence of the edges (r', \sim, s) and (s, \xrightarrow{f}, r') in the reference graph according to the definition of the reference graph. For any reference $r', (s, \wedge, r') \in E_{\tilde{h}_1}$ iff $(s, \wedge, r') \in E_{\tilde{h}_2}$ according to $G_{\perp}^{\tilde{h}_1} \cong G_{\perp}^{\tilde{h}_2}$ where $\wedge \in \{\sim, \xrightarrow{f}\}$. Hence, $\tilde{h}_1 \models s \wedge r' \Leftrightarrow \tilde{h}_2 \models s \wedge r'$, and $\tilde{h}_1 \models r' \wedge s \Leftrightarrow \tilde{h}_2 \models r' \wedge s$, for all $r' \in \mathcal{R}$, and the updates of $\text{UpdHpRel}(r = s)$ and $\text{UpdHpRel}(r = s)$ will be the same in both heaps (i).
 Form Definition 2 (ii) and $\tilde{h}_1 =_{\mathfrak{h}} \tilde{h}_2$, it follows that the primitive fields of s hold the same valuations in both heaps, i.e., $\forall x. \tilde{h}_1 \models s.f_p = x \Leftrightarrow \tilde{h}_2 \models s.f_p = x$, for all $f_p \in \text{PFields}$ (ii). Furthermore, \vec{r} will be set to \vec{s} according to Fig. 2 (iii). From $\tilde{h}_1 =_{\mathfrak{h}} \tilde{h}_2$ and (i)-(iii), it follows that $\tilde{h}'_1 =_{\mathfrak{h}'} \tilde{h}'_2$.
- (ii) If $\vec{s} = \top$, this means that $s \notin N_{\tilde{h}_i}, i \in \{1, 2\}$ according to the definition of reference graphs, and Proposition 1. If $\vec{r} = \top$, then $r \notin N_{\tilde{h}_i}$ and subsequently $G_{\perp}^{\tilde{h}_i}$ will not be updated by $\text{UpdHpRel}(r = s), i \in \{1, 2\}$, i.e., $G_{\perp}^{\tilde{h}'_i} \cong G_{\perp}^{\tilde{h}_i}$ (i). If $\vec{r} = \perp$, then r will be removed from $G_{\perp}^{\tilde{h}'_i}, i \in \{1, 2\}$, because all its connections will be updated to the corresponding ones of s according to $\text{UpdHpRel}(r = s)$. Since, $G_{\perp}^{\tilde{h}_1} \cong G_{\perp}^{\tilde{h}_2}$, it's obvious that the updated graphs will remain isomorphic (ii). Furthermore, \vec{r} will be \vec{s} according to Fig. 2 (iii). From $\tilde{h}_1 =_{\mathfrak{h}} \tilde{h}_2$ and (i)-(iii), it follows that $\tilde{h}'_1 =_{\mathfrak{h}'} \tilde{h}'_2$.

For the proof of field load $r = s.f_r$, we perform a case analysis on \vec{s} , and the proof is similar to the one above. For the case $r = \text{null}$, \vec{r} will be updated to \perp and r will be included in $G_{\perp}^{\tilde{h}'_i}$, as an isolated node, according to $\text{UpdHpRel}(r \leftarrow \cdot)$ and $(r = \text{null})(i)$. Hence, from (i) and the fact that $G_{\perp}^{\tilde{h}_1} \cong G_{\perp}^{\tilde{h}_2}$, it obviously follows that $G_{\perp}^{\tilde{h}'_1} \cong G_{\perp}^{\tilde{h}'_2}$.

We should also consider any mutation operation mu . To prove these, we do a case analysis on mu . We prove the case of a reference field store that is more complex, i.e., $r.f_r = s$ where $\vec{s} \sqsubseteq l$. For any reference x s.t. $x \sim r \vee x \xrightarrow{*} r$ holds, \vec{x} will be upgraded by l . Two cases can happen:

- $l = \perp$ From $\vec{s} \sqsubseteq l$, it follows that $\vec{s} = \perp$ and $s \in G_{\perp}^{\tilde{h}_i}$. Since, $l = \perp$, this means that no security level will be updated, i.e., the nodes of $G_{\perp}^{\tilde{h}'_i}$ and $G_{\perp}^{\tilde{h}_i}$ will be the same (i).
 If $\vec{r} = \perp$, we can show that the updates of $\text{UpdHpRel}(r.f_r = s)$ and $\text{UpdHpRel}(r.f_r = s)$ will be the same in both concrete heaps (iii), similar to the case of reference assignment. Hence, from (i) and (iii), we can conclude that $G_{\perp}^{\tilde{h}'_1} \cong G_{\perp}^{\tilde{h}'_2}$.
 If $\vec{r} = \top$, then $r \notin G_{\perp}^{\tilde{h}_i}$ according to the definition of reference graph. From Proposition 1, it follows that $\vec{x} \sqsupseteq \vec{r}$, and hence, $\vec{x} = \top$ and $x \notin G_{\perp}^{\tilde{h}_i}$ (ii). From (i) and (ii), it follows that $x \notin G_{\perp}^{\tilde{h}'_i}$ and $r \notin G_{\perp}^{\tilde{h}'_i}$, i.e., $x \in G_{\perp}^{\tilde{h}'_i}$ remains unmodified, and consequently $G_{\perp}^{\tilde{h}'_1} \cong G_{\perp}^{\tilde{h}'_2}$.
- $l = \top$ If $l = \top$, then $\vec{x} = \top$ and consequently $x \notin G_{\perp}^{\tilde{h}'_i}$. According to $\text{UpdHpLev}(r, l)$, the level of any reference x that aliases or transitively field-aliases r will be

upgraded to \top , and consequently will not belong to $G_{\perp}^{h'_i}$ anymore. Informally, this means that r and all references that can reach r will be removed from $G_{\perp}^{h_i}$. Since $G_{\perp}^{h_1} \cong G_{\perp}^{h_2}$, we can conclude that this set of references and their incoming/outgoing edges will be the same in both concrete heaps. Hence, removing them from the reference graphs will still lead to isomorphic graphs, i.e., $G_{\perp}^{h'_1} \cong G_{\perp}^{h'_2}$.

Since, no primitive field is updated by a reference field store, the condition (ii) in Definition 2 will still hold for the updated heaps $h'_i, i \in \{1, 2\}$.

The proof for the case of the primitive field store $r.f_p = v$ is similar to that of the reference field store. We prove $G_{\perp}^{h'_1} \cong G_{\perp}^{h'_2}$ by performing case analysis on l . According to $\text{UpdPFields}(op)$, the primitive field f_p of all references aliasing r will be updated to v in both heaps h'_i . This means that for any reference r' that belongs to $G_{\perp}^{h'_i}$, the updates to its primitive field f_p will be the same in both heaps.

B Proof of Noninterference

We use the notation $q(a)$ to show the value of a in the state q , use $q(\vec{h})$ to denote $q(\mathcal{X}_{\vec{h}})$, and $q(\mathbf{h})$ to show $q(\mathcal{X}_{1\mathbf{h}})$. Furthermore, we say $q_1(\vec{h}) \sqsubseteq q_2(\vec{h})$, if and only if for all $r \in R$, $q_1(\vec{r}) \sqsubseteq q_2(\vec{r})$. To prove Theorem 2, we first need to define the notion of compatible states.

Definition 8 (Compatible States). *We say two states q_1 and q_2 are compatible, denoted by $q_1 \approx q_2$, iff*

- (1) $q_1(\ell) = q_2(\ell)$,
- (2) $q_1(\Omega) = q_2(\Omega)$,
- (3) $q_1(V) =_{q_1(\Omega)} q_2(V)$,
- (4) $q_1(\vec{h}) = q_2(\vec{h})$,
- (5) $q_1(\mathbf{h}) =_{q_1(\mathbf{h})} q_2(\mathbf{h})$,

Note that we assume that the information-flow summaries provided by the user are correct, i.e., the summary update preserves the low-equivalence relation in the low context.

Proof. To prove this theorem, we need to find a witnessing bisimulation relation β that witnesses $q_1 \sim_{low} q_2$. We define β as the following and prove that it is a bisimulation relation:

$$\beta = \{ \langle q, q' \rangle \mid q \approx q' \wedge \gamma(q, q') \}$$

where

$$\begin{aligned} \gamma(q, q') &= q(\Upsilon) = q'(\Upsilon) = \text{ff} \wedge \\ & q(pc) = q'(pc) = \perp \wedge q(hr) = q'(hr) = \text{P}_{\perp}. \end{aligned} \quad (2)$$

The initial states q_0 and q'_0 are obviously in the relation β according to Def. 5, $q_0 \models x_{0-all}$ and $q'_0 \models x_{0-all}$. Let $\langle q, q' \rangle \in \beta$. According to Def. 5, we should prove

that, if $q \xrightarrow{\circ}_* t$, then either (a) there exists t' such that $q' \xrightarrow{\circ}_* t'$ and $t\beta t'$, or (b) $q' \xrightarrow{\perp}_*$, and vice versa. If no observation is made from q' , then the case (b) obviously holds. If an observation is made after q' , then the conclusion is followed from Lemma 1 which states that t and t' are in the relation β too.

We call a state q a branching state, iff $q(\ell) = (\text{if } (e) \text{ goto } l; _, _)$. A *branch execution* is an execution $q_0 \xrightarrow{\rho} q_n = q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_n$ in the CDR ρ that starts in a branching state inducing a CDR ρ and exits from ρ , i.e., $q_{n-1}(\ell) = (\text{junc}(\rho), \text{njb})$ and $q_n(\ell) = (\text{junc}(\rho), \text{nb})$.

Lemma 1 (Single Observation). *Let $q_1\beta q_2$. If $q_1 \xrightarrow{\circ}_* q'_1$ and $q_2 \xrightarrow{\circ}_* q'_2$, then $q'_1\beta q'_2$.*

Proof. We prove this by induction on the length of $q_1 \xrightarrow{\circ}_* q'_1$. Since, the execution does not terminate in q_1 , this means $q_1(\ell) \neq (\surd, _)$ and $q_1(\ell) = (a; _, _)$.

Base Case The base case (i.e., the length of one) happens when a is an output statement like $\text{output}_l(y)$.

In case of $\text{output}_l(y)$, \bar{y} can not be high-sensitive according to the invariant $\varphi(\ell)$ in the rule SINK. Hence, according to the conditions (3) and (2) in Def. 8, the observations should be the same in both states. Since it does not update the state, apart from the symbolic location, the target states will remain compatible as well.

Inductive Step Let it hold for all executions $q_1 \xrightarrow{\circ}_* q'_1$ with the length of m or less than m . We should prove the lemma for an execution with the length of $m + 1$.

Let a be a junction point. From $q_1\beta q_2$ and the condition in Eq. 2, it follows that $q_i(\mathcal{Y}) = \text{ff}$ and $q_i(\text{hr}) = \text{P}_\perp, i \in \{1, 2\}$. According to JUNC and M-JUNC, the first transition of M-JUNC is enabled in both states that leads to changing the symbolic location (Note that P_J is obviously the same in both states).

If a is not a junction point, we perform a case analysis on a .

- (i) Let a be an assignment where $q_1 \rightarrow t_1$ and $q_2 \rightarrow t_2$. From Lemma 2, it follows that $t_1\beta t_2$. The conclusion is derived by applying the inductive hypothesis on t_1 and t_2 .
- (ii) Let a be an output statement $\text{output}_l(y)$. Since these states have not been avoided by the security guards, this means that the invariant in the rule SINK holds in $q_i, i \in \{1, 2\}$. However, observations should be produced in the last transition of this execution according to the definition of $\xrightarrow{\circ}_*$. Since, this transition is not the last one, this means that a cannot produce any observation, i.e., the invariant in the rule SINK does not hold and a cannot be an output statement.
- (iii) Let a be goto label . Since, only the semantic location will change to $(\text{target}(\text{label}), \text{njb})$ according to GOTO and M-STM, it is obvious that $t_1\beta t_2$, where $q_1 \rightarrow t_1$ and $q_2 \rightarrow t_2$. The conclusion is derived by applying the inductive hypothesis on t_1 and t_2 .
- (iv) Let a be a method call $r.m'(w)$. The update comprises executing a sequence of non-interfering assignments. We can prove that the two states after applying

each assignment individually will still remain in relation, in a similar fashion to the proof of Lemma 2. Since the assignments by the summary update are non-interfering, it's trivial to show that the relation will be preserved after applying the update. The conclusion is derived by applying the inductive step hypothesis on the states after the update.

- (v) Let a be `if (e) goto label`. From $q_1\beta q_2$ and the condition in Eq. 2, it follows that $pc = \perp$ and $\mathcal{Y} = \text{ff}$ in both states. Two cases can happen:
- If $q_i(\bar{e}) = \perp$, from $q_1\beta q_2$ and the condition (3) in Def. 8, it follows that $q_1 \models e$ iff $q_2 \models e$, i.e., the same branch will be taken from both states. Let them respectively evolve into states t_1 and t_2 . According to M-BRCH and BRANCH, the semantic location will change to the same value in both cases and no other update will be done, i.e., it follows that $t_1\beta t_2$. The conclusion is derived by applying the inductive hypothesis on the executions $t_1 \xrightarrow{o_*} q'_1$ and $t_2 \xrightarrow{o_*} q'_2$.
 - If $q_i(\bar{e}) \not\sqsubseteq q_i(pc)$ holds, this means that $q_i(\bar{e}) = \top$. Let $q_i \rightarrow s_i, i \in \{1, 2\}$ and ρ be the CDR induced by a . According to BRANCH and M-BRCH, $s_i(pc) = q_i(pc) \sqcup q_i(\bar{e}) = \top$. From Lemma 3-1, pc of all states of ρ 's execution is \top . This means that the execution in ρ will lead to no observation, as the invariant in the rule SINK would have been violated, since $pc \neq \perp$. This execution will finally lead to an observation, according to the premises, i.e., its pc will become \perp . According to Lemma 3-1, pc will reset to $q_i(pc)$ in the junction, i.e., ρ 's junction will be visited and $q_i \xrightarrow{\rho} t_i$ for some $t_i, i \in \{1, 2\}$. According to Lemma 6, it follows that $t_1\beta t_2$. The conclusion is derived by applying the inductive hypothesis on t_1 and t_2 .

Lemma 2 (Assignment Preserves Relation). *Let $q_1\beta q_2$ where $q_i(\ell) = (a; _, _), i \in \{1, 2\}$ and a is an assignment. If $q_1 \rightarrow q'_1$ and $q_2 \rightarrow q'_2$, then $q'_1\beta q'_2$.*

Proof. Case analysis on a and $i \in \{1, 2\}$.

- (i) $v = e$: From $q_1\beta q_2$, the conditions (2) and (4) in Def. 8 and the rule M-ASSIGN in Fig. 8, $q_i(\Omega)$ will be updated to the same value of $q_i(\bar{e}) \sqcup q_i(pc)$, i.e., the conditions (2) and (4) in Def. 8 will hold. Note that the heap will not be affected.
- If $q_i(\bar{e}) = \perp$, this means that e contains no high-sensitive variables. Since, the low-sensitive variables have the same values in both states based on Def. 8-(3) and Def. 8-(5), it follows that v will be updated to the same value in both cases, and the conditions (3) and (5) of Def. 8 will still hold for q'_1 and q'_2 . If $q_i(\bar{e}) = \top$, $q'_i(v)$ might differ. However, the conditions (3) and (5) of Def. 8 will still hold for q'_1 and q'_2 according to Def. 4, as $q'_i(\Omega) = \top$.
- The variables pc, hr, \mathcal{Y} will remain unchanged, i.e., Eq. 2 will still hold. According to the semantics rule M-ASSIGN, ℓ will be updated to the same value in both states, i.e., the condition (1) will hold. Hence, all the conditions in Def. 8 will hold, and consequently, $q'_1\beta q'_2$.
- (ii) $v = r.f_p$: Similar to the case (i), we can prove that all the conditions in Def. 8 will hold for the pair of q'_1 and q'_2 .

(iii) For the cases of $r = s$, $r = \mathbf{new} \ c$, $s = r.f_r$, $r.f_r = s$, $r.f_p = e$ and $r = \mathbf{null}$, we can show that the condition (1) and Eq. 2 will hold in the new states similar to the first case. The conditions (2) and (3) are followed from the fact that local primitive variables are not modified.

If $r = r'$, $s = r.f_r$, $r.f_p = e$ or $r = \mathbf{null}$, the conditions (2)-(5) are followed from Theorem 1. If $r = \mathbf{new} \ c$, then l in Def. 3 will be set to pc according to the rule ASSIGN. If $s = r.f_r$, then $l = [\bar{s} \sqcup \vec{s}]_{\mathcal{Y}}$ according to T_a in ASSIGN. Since, $q_i(pc) = \perp$ and $\mathcal{Y} = \perp$, then $l = q_i(\bar{s}) \sqcup q_i(\vec{s}) \sqcup pc$ based on the definition of $[\cdot]_{\mathcal{Y}}$, i.e., $l \sqsubseteq q_i(\bar{s}) \sqcup q_i(\vec{s})$ (i). Similarly, if $r.f_p = e$, then $l = [\bar{e}]_{\mathcal{Y}}$, and subsequently, $l \sqsubseteq q_i(\bar{e})$ (ii). From (i), (ii) and Theorem 1, the conditions (2)-(5) are followed. Hence, all conditions hold, we conclude that $q'_1 \beta q'_2$.

Lemma 3 (High-Context Branch Invariants). *Let $q_0 \xrightarrow{\rho} (q_n)$ be a (possibly finite) branch execution where $q_1(pc) \sqcup q_1(\bar{e}) = \top$. For $1 \leq i < n$, the following properties hold:*

1. $q_i(pc) = \top$ and $q_i(pc') = q_0(pc)$ (and $q_n(pc) = q_0(pc)$).
2. $q_i(hr) = P_\rho$.
3. If $q_0 \xrightarrow{\rho} q_n$, then
 - (a) there exists $j < n$ such that (i) for all $0 < k \leq j$, $q_k(\mathbf{h}') = q_0(\mathbf{h})$ and $q_k(\mathcal{Y}) = \perp$, (ii) $q_{j+1}(\mathbf{h}) = q_0(\mathbf{h})$, (iii) for all $j < k < n$, $q_k(\mathbf{h}') = q_j(\mathbf{h})$ and $q_k(\mathcal{Y}) \neq \perp$, and
 - (b) $q_n(hr) = P_\perp$, and $q_n(\mathcal{Y}) = \perp$.

Proof. Trivial by induction on i and then a case analysis on the statements. In the proof of item 3a, we use contradiction to show that j exists.

Lemma 4 (Updates in High-Context). *Let $q_0 \xrightarrow{\rho} q_n$ where $q_1(pc) \sqcup q_1(\bar{e}) = \top$. For all $0 \leq i < n$, $i \neq j$ where $0 < j < n$, $q_j(\mathcal{Y}) = \perp$, $q_j(hr) = P_\rho$ and $q_j(\ell) = (_, \mathbf{njb})$, the following properties hold:*

1. For all $a \in P$, $q_i(\bar{a}) \sqsubseteq q_{i+1}(\bar{a})$, and for all $r \in R$, $q_i(\bar{r}) \sqsubseteq q_{i+1}(\bar{r})$ and $q_i(\vec{r}) \sqsubseteq q_{i+1}(\vec{r})$.
2. For all $a \in P \cup R$, $q_{i+1}(a) = q_i(a)$ if $q_{i+1}(\bar{a}) = \perp$.
3. For all $a \in P \cup R$, $q_n(a) = q_0(a)$ if $q_n(\bar{a}) = \perp$.
4. For all $r, s \in R$ where $q_i(\vec{r}) = \perp$, (i) $q_i(r \sim s) = q_0(r \sim s)$, and (ii) $q_i(r \xrightarrow{*} s) = q_0(r \xrightarrow{*} s)$.

Proof. By induction on i and then a case analysis on the statements. Informally, the security type of any variable updated in the execution will be at least upgraded by pc . Since, $pc = \top$ according to Lemma 3-1, then the label of no variable will be downgraded. Note that downgrading is not done in the junction state q_j where the junction is visited in the nominal mode and leads to applying the updates by Start-ua that updates \mathbf{h} to \mathbf{h}' .

Lemma 5 (Equal Typing Environment in High-Context). *Let $q_0 \xrightarrow{\rho} q_n$ and $t_0 \xrightarrow{\rho} t_m$ be two executions where $q_0 \beta t_0$, and $q_0(pc) \sqcup q_0(\bar{e}) = \top$. It holds that*

(i) for all $x \in P \cup R$, $q_n(\bar{x}) = \perp$ iff $t_m(\bar{x}) = \perp$, and (ii) for all $r \in R$, $q_n(\vec{r}) = \perp$ iff $t_m(\vec{r}) = \perp$.

Proof. We prove (i) by contradiction. Let $q_n(\bar{x}) = \perp$ and $t_m(\bar{x}) = \top$ for some x . From $q_0 \beta t_0$ and the condition (4) in Def. 8, it follows that $q_0(\bar{x}) = t_0(\bar{x})$. If $q_0(\bar{x}) = t_0(\bar{x}) = \top$, then we can conclude that $q_0(\bar{x}) \sqsubseteq q_n(\bar{x})$ using Lemma 4-1 and the transitivity property of \sqsubseteq , i.e., $q_n(\bar{x}) = \top$ that contradicts our assumptions and proves the conclusion.

If $q_0(\bar{x}) = t_0(\bar{x}) = \perp$, from $t_m(\bar{x}) = \top$, it follows that a statement a has upgraded the level of x . Based on the definition of valid executions that requires execution of all branches in the upgrade analysis mode, a should have been executed in $q_0 \xrightarrow{\rho} q_n$ as well. We show that $q_n(\bar{x}) = \top$ by performing a case analysis on a . The statement a cannot be a (conditional) `goto`, `output`, junction or \surd , as none can update \bar{x} .

- Let a be an assignment. If a is not a field load, since a updates the security type of x in $t_0 \xrightarrow{\rho} t_m$, then $a = \langle\langle x = x' \rangle\rangle$ for some x' according to ASSIGN. If $\mathcal{Y} = \text{tt}$, then \bar{x} is upgraded by pc by the statement a in $q_0 \xrightarrow{\rho} q_n$ too according to ASSIGN, i.e., $q_i(\bar{x}) = q_{i-1}(\bar{x}) \sqcup q_{i-1}(pc)$. From Lemma 3-1, $q_{i-1}(pc) = \top$, and consequently $q_i(\bar{x}) = \top$. Using Lemma 4-1, it follows that $q_i(\bar{x}) \sqsubseteq q_n(\bar{x})$ i.e., $q_n(\bar{x}) = \top$ that contradicts our assumption.

To prove (ii), we induct on the number of CDRs of ρ .

Base Case According to the rules M-BRCH and BRANCH and the fact that the CDRs have unique junctions, the function End-ua applies the final updates $\text{BulkUpgr}_{\mathfrak{h}' \leftarrow \mathfrak{h}}$ to the heap in the states q_{n-1} and t_{m-1} , which is equal to $\text{CopyRels}_{\mathfrak{h}' \leftarrow \mathfrak{h}} \sqcup \text{RstrLev}_{\mathfrak{h}' \leftarrow \mathfrak{h}'}$ in Fig. 2. From Lemma 3-3a, it follows that $q_{n-1}(\mathfrak{h}') = q_j(\mathfrak{h})$ where j is ρ 's junction visited in the nominal mode (Note that ρ has no inner CDR in the base case). Therefore, the updates to the heap by End-ua will be equivalent to $\text{CopyRels}_{q_{n-1}(\mathfrak{h}) \leftarrow q_j(\mathfrak{h})} \sqcup \text{RstrLev}_{q_{n-1}(\mathfrak{h}) \leftarrow q_j(\mathfrak{h})}$. Using Lemma 3-3a-(ii), and Lemma 4-1 and the transitivity property of \sqsubseteq , we can show that $q_0(\vec{\mathfrak{h}}) \sqsubseteq q_{n-1}(\vec{\mathfrak{h}})$ and $q_0(\vec{\mathfrak{h}}) \sqsubseteq q_j(\vec{\mathfrak{h}})$. Consequently, according to the definition of $\text{RstrLev}_{q_{n-1}(\mathfrak{h}) \leftarrow q_j(\mathfrak{h})}$, we can conclude that no downgrading will be done, i.e., $q_0(\vec{\mathfrak{h}}) \sqsubseteq q_n(\vec{\mathfrak{h}})$ (I).

We prove the base case by contradiction. Let $q_n(\vec{r}) = \perp$ and $t_m(\vec{r}) = \top$ for some r . From $q_0(\vec{\mathfrak{h}}) = t_0(\vec{\mathfrak{h}})$, it follows that $q_0(\vec{r}) = t_0(\vec{r})$. If $q_0(\vec{r}) = t_0(\vec{r}) = \top$, then we can conclude that $q_0(\vec{r}) \sqsubseteq q_n(\vec{r})$ using (I), i.e., $q_n(\vec{r}) = \top$ that contradicts our assumptions and proves the conclusion. If $q_0(\vec{r}) = t_0(\vec{r}) = \perp$, from $t_m(\vec{r}) = \top$, it's followed that a statement a has upgraded the level of r in the execution $t_0 \xrightarrow{\rho} t_m$. According to the definition of valid executions, this statement should have been executed by $q_0 \xrightarrow{\rho} q_n$ too. According to the definition of $\text{RstrLev}_{q_{n-1}(\mathfrak{h}) \leftarrow q_j(\mathfrak{h})}$ and the reflexivity property of \sim (i.e., $s \sim^{q_{n-1}(\mathfrak{h})} s$), we can conclude that $q_n(\vec{x}^{q_{n-1}(\mathfrak{h})}) = q_{n-1}(\vec{x}^{q_{n-1}(\mathfrak{h})}) \sqcup q_{n-1}(\vec{x}^{q_{n-1}(\mathfrak{h}')})$. This means that the label of x should be \perp in both $q_{n-1}(\vec{x}^{q_{n-1}(\mathfrak{h})})$ and $q_{n-1}(\vec{x}^{q_{n-1}(\mathfrak{h}')})$.

Let a be a statement that modifies the level of x in $t_0 \xrightarrow{\rho} t_m$. The only operation that can manipulate \vec{x} should be a field store operation according to T_a in the rule ASSIGN (or other operations that might include field restore operations). Let a be a field store $r.f_r = s$ executed in a state $t_j, 0 \leq j < m$ and led to upgrading \vec{x} . According to the semantics, either $t_j(x \sim r)$ or $t_j(x \overset{*}{\rightarrow} r)$ should hold (II). Based on Lemma 4-4, $t_j(x \sim r) = t_0(x \sim r)$ if $x \sim r \in \mathbb{V}_{\mathcal{R}}$ and $t_j(x \overset{*}{\rightarrow} r) = t_0(x \overset{*}{\rightarrow} r)$ if $x \overset{*}{\rightarrow} r$ (III). As mentioned earlier, the statement a is executed in the execution $q_0 \xrightarrow{\rho} q_n$ too, say in a state $q_k, 0 \leq k < n$. Similar to the case of first execution, we can show that $q_k(x \sim r) = q_0(x \sim r)$ if $x \sim r \in \mathbb{V}_{\mathcal{R}}$ and $q_k(x \overset{*}{\rightarrow} r) = q_0(x \overset{*}{\rightarrow} r)$ if $x \overset{*}{\rightarrow} r$ (IV). From III, IV, $q_0 \beta t_0$ and Def. 8-(5), it follows that $q_k(x \sim r) = t_j(x \sim r)$ if $x \sim r \in \mathbb{V}_{\mathcal{R}}$ and $q_k(x \overset{*}{\rightarrow} r) = t_j(x \overset{*}{\rightarrow} r)$ if $x \overset{*}{\rightarrow} r$ (V). From (II) and (V), it follows that $q_k(x \sim r)$ or $q_k(x \overset{*}{\rightarrow} r)$ holds. Since, $q_k(pc) = \top$, it follows that $l = \top$ in ASSIGN, that leads to upgrading $\vec{x} = \top$. Since no downgrading is done in non-junction states according to Lemma 4-1, this means that $q_{n-1}(\vec{x}^{q_{n-1}(\mathbf{b})}) \sqcup q_{n-1}(\vec{x}^{q_{n-1}(\mathbf{b}')})) = \top$ which contradicts our assumption and proves the conclusion, i.e., $q_n(\vec{x}) = \top$.

The case of the primitive field store operation is proven similarly.

Inductive Step Straightforward.

Lemma 6 (High-Context Preserves Relation). *Let $q_1 \beta q_2$ where (i) $q_i(\ell) = \text{if } (e) \text{ goto label; } _ , _ , i \in \{1, 2\}$, and (ii) $q_1(pc) = \perp$ and $q_1(\bar{e}) = \top$. If $q_1 \xrightarrow{\rho} q'_1$ and $q_2 \xrightarrow{\rho} q'_2$, then $q'_1 \beta q'_2$.*

Proof. We show that all conditions in Def. 8 and Eq. 2 hold for the pair (q'_1, q'_2) .

- $q'_1(\ell) = q'_2(\ell)$ follows from the definition of valid executions and the fact that each CDR has a unique junction.
- $q'_1(\mathcal{Y}) = q'_2(\mathcal{Y}) = \text{ff}$, $q'_1(pc) = q'_2(pc) = \perp$ are $q'_1(hr) = q'_2(hr) = \text{P}_{\perp}$ are followed from Lemma 3.
- $q'_1(\Omega) = q'_2(\Omega)$ is followed from Lemma 5.
- According to Lemma 4-3, it's followed that for all $a \in P \cup R$, $q'_i(a) = q_i(a)$, if $q'_i(\bar{a}) = \perp$ (i). From $q_1 \beta q_2$, Def. 8-(3), Def. 8-(5), it follows that $q_1(a) = q_2(a)$, if $q'_i(\bar{a}) = \perp$ (ii). From (i), (ii) and Lemma 5, we can conclude that for all $a \in P \cup R$, $q'_1(a) = q'_2(a)$, if $q'_1(\bar{a}) = \perp$. Consequently, $q'_1(\mathcal{V}) =_{q'_1(\Omega)} q'_2(\mathcal{V})$ and $q'_1(\mathbf{h}) =_{q'_1(\Omega)} q'_2(\mathbf{h})$ are followed.