A Theorem Prover Backed Approach to Array Abstraction*

Nathan Wasser and Richard Bubel

Technische Universität Darmstadt, Darmstadt, Germany
{wasser, bubel}@informatik.tu-darmstadt.de

Abstract

We present an extension to an on-demand abstraction framework, which integrates deductive verification and abstract interpretation. Our extension allows for a significantly higher precision when reasoning about programs containing arrays. We demonstrate the usefulness of our approach in the context of reasoning about secure information flow. In addition to abstracting arrays that may have been modified, our approach can also keep full precision while adding additional information about array elements which have been only read but not modified.

1 Introduction

Verification and testing have become the bottleneck in large software projects. In fact, rather than the verification itself, it is the specification that is the real bottleneck. In most non-trivial areas specification must be done for the most part manually by highly paid and scarce individuals. Therefore any approach that automates at least part of the specification process can be increasingly useful in saving time and money and reducing the bottleneck for specification somewhat. Especially in regards to the automatic generation of loop invariants there are both built-in problems with automation in that there are obvious limits to any algorithmic approach as shown by the undecidability theorem, as well as merely the lack of good tools in this area.

We present an extension to an on-demand abstraction framework [4], which integrates deductive verification and abstract interpretation. Our extension allows for a significantly higher precision when reasoning about programs containing arrays. In addition to abstracting arrays that may have been modified, our approach can also keep full precision while adding additional information about array elements which have been only read but not modified.

We demonstrate the usefulness of our approach in the context of reasoning about secure information flow. Programs that have secure information flow do not leak secret information to publicly accessible channels.

Outline. Section 2 introduces the logical framework with on-demand abstraction and provides a brief overview of information flow security. In Section 3 we present our extension, which we then utilize for information flow analysis of program in Section 4. We conclude and outline future work in Section 5.

Related Work. Our work builds upon [4] where the logical framework is introduced. We extended the approach from an academic toy language to a subset of sequential Java and are implementing tool support based on KeY. A first version of the tool is available at `www.se.tu-darmstadt.de/research/projects/albia/download/`. Our approach to reasoning about information flow properties differs in the tracking of implicit dependencies and avoids complicated non-standard rules for branching program statements.

*The work has been funded by the DFG priority program 1496 "'Reliably Secure Software Systems'"

There are several approaches to the generation of loop invariants including but not limited to arrays [3, 5, 9]. Our approach is deeply integrated into a fully precise program logic with on-demand abstraction and can thus maintain a high precision and produce strong invariants.

2 Background

2.1 Information Flow and Non-Interference

Securing data in computing systems is a challenging endeavor. Allowing information disclosure, while limiting it in such a way as to deny secret information from being publicly observed can be a tricky task. Information flow analysis usually assigns each hardware input, method parameter, or similar source a security level, typically either *low* or *high*. Likewise all sinks – be they hardware outputs, return values or allocated memory areas – are assigned a security level. All flows may then be analyzed to see, for example, if high inputs could be flowing to low outputs. Allowed and forbidden flows are specified by information flow policies like non-interference or delimited information release. The most strict policy is non-interference which states no information at all may leak from high to low security locations. In other words, an attacker is not able to distinguish multiple computations by observing their low outputs (and low inputs), if they differ only in their high inputs.

In general this policy is often too strict and requires a form of declassification like delimited information release which allows a certain well-specified amount of secret information to be leaked. For instance, a password checker leaks whether the entered password is correct or incorrect, which is the expected behaviour. One well-known way to enforce information flow policies is through a security type system. Here any variable within the program is assigned a certain security level, which allows it to contain only information of the specified or a lower security level. While this is a sound approach, ensuring that all well-typed programs obey the policy, it may fail to recognize actually secure programs. Rewriting a program which is not well-typed, but has no improper information flows can be tedious and non-trivial. Typically, type-based systems have a high abstraction level which allows them to stay fully automatic, but comes at the cost of an increased number of false positives. Other approaches as the one presented in this paper use a logic formalisation of secure information flow [6, 7, 1], but differ in the degree of automation.

Information flow dependencies can be both direct, for example, an assignment $x = y$ flows information from y to x, or indirect as in executing different code based on a certain condition: The statement `if` (b) `then` t `else` e flows information about b into both statements t and e.

Example 1. *In the following example programs,* h *and* l *are program variables, where* h *has security level* High *and* l *security level* Low*. A program is considered* secure *if an attacker who reads the final values of the* Low *variables cannot infer any information about the initial values of the* High *variables.*

1. l=h *is obviously* insecure*, because information flows directly from* h *to* l*.*
2. **if** (h>0) {l=1} **else** {l=2} *is also* insecure*, because information about the sign of the initial value of* h *flows indirectly to* l*.*
3. **if** (l>0) {h=1} **else** {h=2} *is* secure*, because the value of* l *is not touched at all.*
4. **if** (h>0) {l=1} **else** {l=2}; l=3 *is* secure*, because the final value of* l *is always* 3*, independently of the initial value of* h*.*
5. h=0;l=h *is* secure*, because the final value of* l *is always* 0*.*

6. **if** (h>0) {h=l;l=h} *is* secure*, because the value of* l *is not changed.*
7. **if** (h>0) {l=2;h=1} **else** {l=2;h=2} *is* secure*, because the final value of* l *is always* 2.
8. l=h−h *is* secure*, because the final value of* l *is always* 0.

2.2 Underlying Logic Framework

2.2.1 Dynamic Logic for Java

The logic we use to present our approach on generating array invariants is dynamic logic [10], or more precise, Java Card Dynamic Logic (JavaDL) [2] using the explicit heap model as developed in [13]. Our programming language is sequential Java without dynamic class loading, garbage collection and floats.

The described approach can be easily adapted by any other program logic and calculus, which uses an explicit representation of the symbolic state and to a lesser extent makes use of symbolic execution.

In dynamic logic, programs are first class citizens, i.e., programs occur syntactically as part of the formulas and not in an encoded form. Dynamic logic (and thus JavaDL) is a first-order logic with two additional modalities $\langle\cdot\rangle\cdot$ (diamond) and $[\cdot]\cdot$ (box). The first argument takes a sequence of executable statements and the second argument an arbitrary dynamic logic formula. At this stage we are not concerned with termination of programs and therefore we restrict ourselves to the box modality, which is sufficient to encode partial correctness.

Intuitively, the formula $[\mathtt{s}]post$ expresses that if program `s` terminates then in its final state formula *post* holds. Dynamic logic is closed under quantification and subsumes Hoare logic. The Hoare triple $\{P\}\mathtt{s}\{Q\}$ is equivalent to the DL formula $P \rightarrow [\mathtt{s}]Q$.

Example 2. *The formula* $\mathtt{x} >= 0 \rightarrow [\mathtt{while(x>0)\ \{\ x--;\ \}}]\mathtt{x} \doteq 0$ *means that if the program is started in a state where* `x` *is greater than* 0 *and it terminates then in its final state* $\mathtt{x} \doteq 0$ *holds.*

The Java heap is modelled as a datatype `Heap` axiomatized as the theory of arrays containing the following functions and axioms:

$$\mathtt{store} : \mathtt{Heap} \times \mathtt{Object} \times \mathtt{Field} \times \mathtt{Any} \rightarrow \mathtt{Heap}$$
$$\mathtt{select} : \mathtt{Heap} \times \mathtt{Object} \times \mathtt{Field} \rightarrow \mathtt{Any}$$
$$\mathtt{select}(\mathtt{store}(h,u,g,z),o,f) \rightsquigarrow \begin{cases} z & , \text{ if } u = o \text{ and } g = f \\ \mathtt{select}(h,o,f) & , \text{ otherwise} \end{cases}$$

An array is a special kind of object which consists of a `length` field and infinitely many fields $\mathtt{arr}(0), \mathtt{arr}(1), \ldots$ with `arr` being a function mapping each integer to a unique field.

JavaDL declares a global program variable `heap` on which Java programs operate, i.e., read and write. For ease of reading we use $o.f$ and $a[i]$ instead of $\mathtt{select}(\mathtt{heap}, o, f)$ resp. $\mathtt{select}(\mathtt{heap}, a, \mathtt{arr}(i))$.

To keep track of state changes (local variable or heap changes) JavaDL uses updates, which can be thought of as explicit substitutions. Let x denote a program variable and t a term of compatible type. An elementary update $x := t$ has the same semantics as an assignment where the right-hand side is side-effect free. Elementary updates u_i can be composed to parallel updates $u_1 \| \ldots \| u_n$ which are executed simultaneously. Conflicts, i.e., if the same variable is assigned different values in a parallel update, are resolved using a last-one-wins conflict resolution.

Updates can be applied to a formula or a term $\{x := t\}\ \xi$ resulting in a new formula resp. term.

Example 3. *To clarify the semantics and usage of updates we give some small examples:*

- *evaluating the formula* $\{\mathtt{i} := \mathtt{i} + 1\}\ (\mathtt{i} > 0)$ *in a state* s *holds iff* $i > 0$ *holds in a state* s' *which coincides with* s *except for the value of* i *which is* $s'(i) = s(i) + 1$
- *evaluating* $\{\mathtt{i} := \mathtt{j} \,||\, \mathtt{j} := \mathtt{i}\}\ \phi$ *in a state* s *is the same as evaluating* ϕ *in a state* s' *which coincides with* s *except that the values of the program variables* i *and* j *are exchanged. Note, this is only possible in this way as parallel updates are applied simultaneously and do not influence each other, in other words, the right-hand side of the elementary updates in a parallel update are evaluated in the pre-state (here:* s*).*
- *the parallel update* $\mathtt{x} := 3 \,||\, \mathtt{x} := 5$ *is equivalent to* $\mathtt{x} := 5$ *as in case of conflicts the last assignment wins.*

Each chain of sequential applications of elementary updates $\{u_1\} \ldots \{u_n\}\phi$ can be rewritten into a parallel update. For further details see [12]. Changes to the Java heap (e.g., assigning a value to an array element) are kept track by updating the global `heap` variable accordingly.

2.2.2 Calculus

To prove that a formula is valid we use a Gentzen-style sequent calculus. A *sequent*

$$\psi_1, \ldots, \psi_m \Rightarrow \phi_1, \ldots, \phi_n \text{ has the same meaning as } (\bigwedge_{i \in \{1,\ldots,m\}} \psi_i) \rightarrow (\bigvee_{j \in \{1,\ldots,n\}} \phi_j)$$

The rules of the sequent calculus are of the general form

$$\mathsf{name}\ \frac{\overbrace{seq_1 \ldots seq_n}^{premiss}}{\underbrace{seq}_{conclusion}}$$

A proof in a sequent calculus is a tree constructed by a sequence of rule applications where each node is labelled with a sequent. For each inner node there is a calculus rule such that the conclusion matches the sequent in the node and the sequents of the children match the instantiated premisses of the rule. A branch is closed if the last applied rule was an axiom rule, i.e., a rule with an empty premiss. A proof is closed if all branches are closed.

The sequent calculus realizes a symbolic interpreter for Java programs. Most of the rules match on the first active statement of a program, i.e., the statement an interpreter would execute next. One example of such a rule is the conditional rule:

$$\frac{\Gamma, b \Rightarrow [p; r]\phi, \Delta \qquad \Gamma, \neg b \Rightarrow [q; r]\phi, \Delta}{\Gamma \Rightarrow [\mathtt{if\ (}b\mathtt{)\ \{}p\mathtt{\}\ else\ \{}q\mathtt{\};}\ r]\phi, \Delta}$$

where Γ, Δ stand for (possibly empty) sets of formulas. Rules are applied in reverse order, i.e., the conclusion is matched against the sequent of an open goal (leaf node). If a match is possible and the rule is applied, the leaf becomes an inner node of which each child corresponds to one sequent of the rule's premisses. For instance, the above rule causes the proof to split into two branches. The left branch assumes that the guard of the conditional statement is true. Here, we have to show that after execution of the *then* branch of the conditional and the rest of the program, we are in a state in which formula ϕ holds. The right branch is concerned with the analogue case where the guard is assumed to be false.

The advantage of symbolic execution is that during verification we follow the normal program control flow. In contrast to a Dijkstra-style weakest precondition computation approach [8], which reasons backwards through the program, we achieve a more natural forward style of reasoning. For this the assignment rule is crucial. We present here the assignment rules for local variables and for array elements:

$$\mathsf{assign}_{loc}\ \frac{\Gamma \Rightarrow \{x := e\}\ [r]\phi, \Delta}{\Gamma \Rightarrow [x\texttt{=}e\texttt{;}\ \ r], \Delta\phi} \qquad \mathsf{assign}_{arr}\ \frac{\Gamma \Rightarrow \{\texttt{heap} := \texttt{store}(\texttt{heap}, a, \texttt{arr}(i), e)\}\ [r]\phi, \Delta}{\Gamma \Rightarrow [a\texttt{[}i\texttt{]=}e\texttt{;}\ \ r]\phi, \Delta}$$

where x, a, i denote program variables and e a side-effect free expression. The assignment rule for local variables turns the assignment directly into an elementary update, while the array assignment rule assigns the global `heap` variable a new heap which coincides with the old one except for the value of $a[i]$.

During symbolic execution the program is stepwise decomposed and updates are accumulated in front of the modality representing the effect of the program. Once the program has been completely executed and the updates have been applied (similar to substitutions) on the formulas, only first-order goals remain which can be proven as usual.

For a loop the simplest approach is to unwind it (note, the version below is simplified ignoring possible **`break`**s or **`continue`**s):

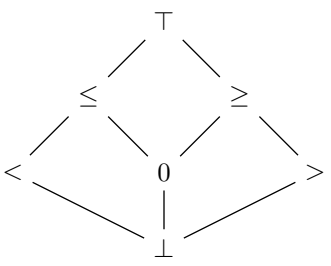
$$\mathsf{loopUnwind}\ \frac{\Gamma \Rightarrow [\texttt{if (}g\texttt{) \{ p; while (}g\texttt{) \{}p\texttt{\}\}; }\ r]\varphi, \Delta}{\Gamma \Rightarrow [\texttt{while (}g\texttt{) \{}p\texttt{\}; }\ r]\varphi, \Delta}$$

Of course, unrolling a loop works only if a fixed bound is known a priori. Otherwise, loop invariants or induction have to be used.

2.2.3 Value-Based Abstraction

Finally, we present briefly how abstract domains and abstract values are represented in our logical framework. We follow previous work of some of the co-authors [4].

A finite abstract domain A is defined as a set of abstract elements $\{\bot, a_1, \ldots, a_n, \top\}$ forming a lattice w.r.t. $\sqsubseteq$. For instance, the sign domain for integers as given in Fig. 1.



$$\begin{aligned}
\gamma(\top) &= \mathbb{Z}\\
\gamma(\leq) &= \{i \in \mathbb{Z} \mid i \leq 0\}\\
\gamma(\geq) &= \{i \in \mathbb{Z} \mid i \geq 0\}\\
\gamma(<) &= \{i \in \mathbb{Z} \mid i < 0\}\\
\gamma(>) &= \{i \in \mathbb{Z} \mid i > 0\}\\
\gamma(0) &= \{0\}\\
\gamma(\bot) &= \{\}
\end{aligned}$$

Figure 1: Abstract domain lattice for sign analysis

One way to support reasoning with abstract domains would be to integrate them into the logic by introducing new types that represent them. This causes major problems as it would require doubling all function symbols, such as addition, and it would be difficult to achieve

well-typedness when assigning program variables abstract domain elements. Instead we went a different route, adding for any abstract element a_i an infinite number of constant symbols $\gamma_{a_i,j}$, where $j \in \mathbb{N}_0$ for which we restrict the interpretation to $\gamma_{a_i,j} \in \gamma(a_i)$, with γ, α being the concretisation and abstraction functions forming a Galois connection. Note, the type of the $\gamma_{a_i,j}$ functions is a concrete domain and the only fact we know is that a $\gamma_{a_i,j}$ belongs to the values represented by a_i. In addition, we add for each abstract element a_i a predicate symbol χ_{a_i} as characteristic function for $\gamma(a_i)$.

2.2.4 Generating Loop Invariants

Being able to represent abstract values in our logical framework, we use the possibility to automatically generate loop invariants. We sketch here only the rough idea, for more details we refer to [4].

As detailed earlier, when verifying a program we start with a Java DL formula like $pre \rightarrow [\mathtt{p}]post$ and try to prove it with our sequent calculus. We start executing `p` symbolically until we reach a loop, i.e., the open goal looks similar to $\Gamma \Rightarrow \{u_0\}[\texttt{while (g)\{bd\}; r}]post, \Delta$.

At that point we have to either provide a loop invariant or perform induction and provide an induction hypothesis. Instead we start to generate an update (called *invariant update*), which describes all possible states that may be reached when leaving the loop. We describe the computation of the invariant update along a small example: Consider the program: `i = 0; while (i<n) { i++; } z = z + i;`. As abstract domain for integers we use the sign domain. We want to prove

$$z_0 \doteq 0 \rightarrow \{\mathtt{z} := z_0\}\,[\texttt{i = 0; while (i<n) \{ i++; \} z = z + i;}]\mathtt{z} \geq 0$$

After some proof steps we arrive at

$$z_0 \doteq 0 \Rightarrow \{\mathtt{i} := 0 \,||\, \mathtt{z} := z_0\}\,[\texttt{while (i<n) \{ i++; \} z = z + i;}]\mathtt{z} \geq 0$$

To continue the verification, we compute the invariant update by unrolling the loop once and symbolically executing the loop body. We compare the update u_0 when entering the loop body with the one after the first iteration u_1. For all locations modified by the loop we abstract their value by the best fitting abstract value a_i which encompasses both values. Here u_0 is $\mathtt{i} := 0 \,||\, \mathtt{z} := z_0$ and u_1 is $\mathtt{i} := 1 \,||\, \mathtt{z} := z_0$. The only changed value is that of `i`. Comparing the value before the loop iteration and afterwards, gives us $\geq$ as the best fitting abstract element of the sign domain. Hence, we replace i in u_1 by a fresh (not yet used) symbol $\gamma_{\geq,3}$, the resulting update u_1' now looks similar to $\mathtt{i} := \gamma_{\geq,3} \,||\, \mathtt{z} := z_0$. We then unroll the loop once again and repeat the process always comparing the update from the n-th iteration with the one from the $n-1$-th iteration until a fixed point is found. The so created update u' describes at least all states that may possibly occur after the loop. We continue the verification with the remaining program under update u'.

In a simplified version taken from [4] the rule to introduce the computed update looks as follows :

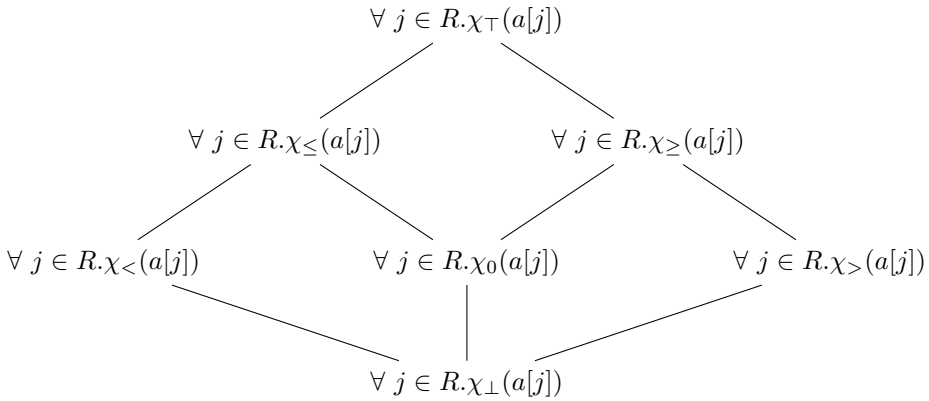
$$\text{invariantUpdate}\ \frac{\begin{array}{l}\Gamma, \{u\}(\bar{x} \doteq \bar{c}) \Rightarrow \exists \bar{\gamma}.\{u'\}(\bar{x} \doteq \bar{c}), \Delta \\ \Gamma, \{u'\}g,\ \{u'\}[\mathtt{p}](\bar{x} \doteq \bar{c}) \Rightarrow \exists \bar{\gamma}.\{u'\}(\bar{x} \doteq \bar{c}), \Delta \\ \Gamma, \{u'\}\neg g \Rightarrow \{\mathcal{U}'\}[r]\varphi, \Delta\end{array}}{\Gamma \Rightarrow \{u\}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta}$$

where u' is the computed abstracted update. We do not go into details here, but basically the first two branches verify that the computed update u' is indeed correct, while in the third branch program execution is continued after the loop and under u'.

3 Generation of array invariants

The value-based abstraction approach works well for primitive types and certain abstract domains for objects, but loses precision quickly in presence of arrays. In this section we refine the approach when dealing with arrays: instead of introducing abstract domains for arrays (e.g., abstracting an array to its length), we utilize the abstract domain of the array elements, formulating invariants that hold for different partitions of the array.

We recall, that for primitive types we use γs which express a constant but freely chosen value within an abstract domain. For example, $\gamma_{>,5}$ is some value, for which one can positively state that it is greater than zero. Besides γs we also have χ-functions which express that the given abstraction is a valid abstraction of the argument. For example, $\chi_{>}(x)$ tells us that x must be greater than zero. A range predicate R decides whether a given index is within the range or not. For this paper we restrict ranges to closed intervals with an according range predicate $R : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. The formula $R(lo, hi, j)$ holds iff $lo \leq j \leq hi$. We write $j \in R(lo, hi)$ instead of $R(lo, hi, j)$ and omit lo and hi when they are clear from the context. For a fixed but arbitrary lo and hi, the χ-classification of array a's elements within that range forms a lattice. Using the sign lattice for integers we get:



In section 2.2.4 we have described the general approach, which requires only the states prior to and after execution of an arbitrary loop iteration in order to calculate the invariant update. As invariant updates alone cannot capture all information, we additionally generate formulas expressing invariants about the array elements which hold for each iteration. To achieve a higher precision we will generate invariants for arrays which either have their elements modified or whose elements influence control flow.

We are concerned with loops that access and possibly modify array elements in a structured way. We therefore analyze two distinct ranges within arrays: The range of elements which have already been processed (R_{acc}) and the range of untouched elements (R_{unmod}). When analyzing a loop we begin with two invariants for each relevant array a: $\forall\ j \in R_{acc}.\chi_{\bot}(a[j])$, which is the neutral element for successively merging additional information about the elements we obtain information about within a loop iteration, and an invariant which states that the elements in R_{unmod} remained unchanged since the start of the loop. This second invariant belongs to a simple lattice which only contains this invariant along with the top and bottom elements for the range R_{unmod}. In case the array accesses cannot be described in the chosen range shape (here: closed intervals), the range designated as untouched may be incorrect, however, the existence of such a top element ensures soundness at the cost of precision. In all other cases this additional information is very often required in order to prove a useful invariant, although usually it is no

longer used after the loop's completion, as in most cases no array elements remain outside of the range R_{acc} upon exiting the loop.

Listing 1: Partial array copy

```
i = 0;
while(i < a.length){
  if (a[i] < 0) {
    a[i] = b[i];
  }
  i = i + 1;
}
```

As an example we want to prove the following sequent:

$$a \neq \textbf{null} \land b \neq \textbf{null} \land a.\texttt{length} \leq b.\texttt{length} \land \forall\ \texttt{int}\ j.\ 0 \leq j < b.\texttt{length} \rightarrow b[j] > 0$$

$$\Rightarrow [\texttt{p}]\ \forall\ \texttt{int}\ j.\ 0 \leq j < a.\texttt{length} \rightarrow a[j] \geq 0,$$

where `p` is the program given in Listing 1. The program partially copies one array into another array. Based on knowledge of the array being copied, as well as local knowledge about the elements of the array being modified, an invariant can be generated classifying the array after the loop.

Using update invariant generation, we infer that i is greater or equal to zero in all loop iterations.

This is done starting at the actual value 0 for the initial state and systematically joining with the new values i can accumulate after each iteration, while ensuring termination by remaining within a fixed-length lattice.

$$\begin{aligned}
\text{Before entering loop: } & i = 0 \\
\text{After first iteration: } & i = \mathit{merge}(0, 1) = \gamma_{\geq,0} \\
\text{After second iteration: } & i = \mathit{merge}(\gamma_{\geq,0}, \gamma_{\geq,0} + 1) = \gamma_{\geq,1} \qquad \text{(fixed point found)}
\end{aligned}$$

The array a will have invariants created for it due to both the control flow depending on its elements as well as the modification thereof. The array b is deemed uninteresting. Based on the generated update information about i – and the additional information that $i < a.\mathit{length}$ (given by the loop guard) at the start of each loop iteration – our starting invariant for the range $R_{acc} = R(0, i-1)$ (heuristically determined) is:

$$\forall\ \texttt{int}\ j.\ 0 \leq j < i \rightarrow \chi_{\perp}(a[j])$$

This invariant is obviously valid only for the empty range, but provides a neutral element as starting point for the joins: After one iteration we join the initial sign lattice element $\perp$ with both the then-branch value of $a[i]$, which is $b[i]$, as well as with the else-branch value of $a[i]$, which while unchanged contains the added information from the branch condition that $a[i] \geq 0$. As $b[i] > 0$ is also assured due to the precondition, the joins result in the new, stricter invariant:

$$\forall\ \texttt{int}\ j.\ 0 \leq j < i \rightarrow \chi_{\geq}(a[j])$$

The second iteration reveals that a fixed point has been found for a, which is strong enough to prove the postcondition.

4 Application to non-interference analysis

To verify whether a program has secure information flow w.r.t. the non-interference property, we make use of the fact that information-flow analysis can be reformulated as an analysis of variable dependencies (see [11]). For any variable `x` we determine the set of variables on whose initial values the final value of `x` can at most depend. We associate security levels with sets of variables. A program adheres to non-interference if the final value of any lower level variable does not depend on a variable with a higher security level.

In order to track dependencies we extend our logic as follows: For each local variable x there is an additional variable x_{dep} containing its dependencies. In the case that a local variable a refers to an array object there is also an additional dependency array a_{depArr} which stores the dependencies of the array elements. Our logic also includes extensions for objects and their fields, but these are skipped for ease of presentation.

Implicit dependencies (caused by branching instructions) are tracked by a global variable `depStack`, which implements a stack of dependency sets. To update the stack, the program is instrumented with special `push`, `pop` and `peek` statements by corresponding calculus rules. For instance, the rule for executing conditional statements is now:

$$\textsf{conditional}\ \frac{\Gamma, b \Rightarrow [\texttt{push(deps(b)} \cup \texttt{peek())}; \texttt{p}; \texttt{pop}(); \texttt{rest}]\phi, \Delta \qquad \textit{analogous for else}}{\Gamma \Rightarrow [\texttt{if(b)\{p\}else\{q\}}; \texttt{rest}]\phi, \Delta}\ ,$$

where $deps(t)$ computes a safe approximation of the dependency set for a side-effect free program expression t.

$$deps(t) := \begin{cases} x_{dep} & \text{if } t \text{ is a local variable } x \\ deps(a) \cup deps(t') \cup a_{depArr}[t'] & \text{if } t = a[t'] \\ deps(a) & \text{if } t = a.\texttt{length} \end{cases}$$

(Note: The approximation of a.length is sound as `length` is a **final** field)

Figure 2: Definition of `deps` (excerpt)

Both explicit and implicit dependencies must be considered whenever an assignment is executed. Here are two of those rules:

$$\textsf{assign}_{local}\ \frac{\Gamma \Rightarrow \{\texttt{x} := \texttt{e} \,||\, \texttt{x}_{\texttt{dep}} := \texttt{deps(e)} \cup \texttt{peek(depStack)}\}\ [\texttt{r}]\phi, \Delta}{\Gamma \Rightarrow [\texttt{x} = \texttt{e}; \texttt{r}]\phi, \Delta}$$

$$\textsf{assign}_{arrayElement}\ \frac{\Gamma \Rightarrow \{\texttt{heap} := \texttt{store(store(heap, a, arr(i), e),}\ \texttt{a}_{\texttt{depArr}}, \texttt{arr(i)}, \texttt{deps(e)} \cup \texttt{deps(i)} \cup \texttt{peek(depStack))}\}\ [\texttt{r}]\phi, \Delta}{\Gamma \Rightarrow [\texttt{a[i]} = \texttt{e}; \texttt{r}]\phi, \Delta}$$

Our intention is to implement a fully automatic verifier for information flow analysis. To achieve automation we use the invariant generation as explained in Section 3. The canonical candidate for an abstract domain for dependency sets is the security lattice given by the application context. In the following example we use the universal lattice (powerset of all program

locations) which coincides with the concrete domain for dependency sets. We analyze the program `p` in Listing 2 w.r.t. the dependencies of the array elements of `a`. Given a set of program variables *high* we aim to prove the following sequent:

$$
\begin{array}{l}
a \neq \textbf{null} \land b \neq \textbf{null} \land a.\texttt{length} \leq b.\texttt{length} \land \\
\forall\ \texttt{int}\ j.\ 0 \leq j < b.\texttt{length} \rightarrow b[j] > 0\ \land\ \forall\ \texttt{int}\ j.\ 0 \leq j < a.\texttt{length} \rightarrow a[j] > 0 \land \\
high \cap deps(a) \doteq \emptyset\ \land\ \forall\ \texttt{int}\ j.\ 0 \leq j < a.\texttt{length} \rightarrow high \cap deps(a[j]) \doteq \emptyset \\
\Rightarrow \\
[\texttt{p}]\forall\ \texttt{int}\ j.\ 0 \leq j < a.\texttt{length} \rightarrow high \cap deps(a[j]) \doteq \emptyset
\end{array}
$$

Listing 2: Dependency tracking

```
i = 0;
while(i < a.length){
  if (a[i] < 0) {
    a[i] = b[i];
  }
  i = i + 1;
}
```

The formula must be valid, as `a` does not initially depend on a high variable and neither do any of its elements. The only point in the program where one of its elements could be assigned a value with a high dependency is within the then-branch of the conditional statement. However, due to the precondition this code is never executed. Figure 3 shows the dependency tracking as performed within the logic during the proof search. Our approach allows to generate a suitable loop invariant which is strong enough to prove the desired property thanks to the advantage of having a theorem prover backing the generation of loop invariants.

Java	Additional steps
i = 0;	$i_{dep} := \emptyset \cup depStack$
while(i < a.length) {	$depStack := push(peek(depStack) \cup i_{dep} \cup a_{dep}, depStack)$
if (a[i] < 0) {	$depStack := push(peek(depStack) \cup a_{dep} \cup i_{dep} \cup a_{dep}[i], depStack)$
a[i] = b[i];	not executed, as in this branch the preconditions are contradictory: $a[i] < 0 \land\ \forall\ \texttt{int}\ j.\ 0 \leq j < a.\texttt{length} \rightarrow a[j] > 0$
}	$depStack := pop(depStack)$ – end of if-statement
i = i + 1;	$i_{dep} := i_{dep} \cup depStack$
}	$depStack := pop(depStack)$

Figure 3: Dependency tracking as performed during proof search

5 Conclusion and Future Work

We extended an abstraction on-demand framework with the ability to maintain a high precision when dealing with arrays. We applied the approach to information-flow analysis. A first implementation is available at `www.se.tu-darmstadt.de/research/projects/albia/download/`. We are currently extending the maturity of the tool and the coverage of the supported sequential Java fragment. Further, we investigate improvements to the loop invariant generation by allowing range predicates for more and different shaped structures. A goal is to be able to deal with array initialisation as found in cryptographic software which can be non-contiguous.

Acknowledgements.

We thank Eduard Kamburjan for help with the implementation and fruitful discussions.

References

[1] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, CSFW-17,Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[2] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.

[4] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madeleine, editors, *Revised Lectures, 7th International Symposium on Formal Methods for Components and Objects (FMCO 2008)*, volume 5751 of *Lecture Notes in Computer Science*, pages 247–277. Springer-Verlag, 2009.

[5] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 105–118. ACM, 2011.

[6] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[7] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.

[8] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[9] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In Jens Palsberg and Martn Abadi, editors, *POPL*, pages 338–350. ACM, 2005.

[10] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[11] Sebastian Hunt and David Sands. On flow-sensitive security types. In *33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 79–90. ACM Press, 2006.

[12] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer-Verlag, 2006.

[13] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.