Hybrid Information Flow Analysis for Programs with Arrays

Gergö Barany*
CEA, LIST, Software Reliability Laboratory
F-91191 Gif-sur-Yvette Cedex, France
gergo.barany@cea.fr

Information flow analysis checks whether certain pieces of (confidential) data may affect the results of computations in unwanted ways and thus leak information. Dynamic information flow analysis adds instrumentation code to the target software to track flows at run time and raise alarms if a flow policy is violated; hybrid analyses combine this with preliminary static analysis.

Using a subset of C as the target language, we extend previous work on hybrid information flow analysis that handled pointers to scalars. Our extended formulation handles arrays, pointers to array elements, and pointer arithmetic. Information flow through arrays of pointers is tracked precisely while arrays of non-pointer types are summarized efficiently.

A prototype of our approach is implemented using the Frama-C program analysis and transformation framework. Work on a full machine-checked proof of the correctness of our approach using Isabelle/HOL is well underway; we present the existing parts and sketch the rest of the correctness argument.

Keywords. information flow, non-interference, termination insensitive non-interference, hybrid program analysis, formal proof

1 Motivation

Information flow analysis is the study of how pieces of confidential data propagate through programs and affect computations. Typically one wishes to enforce a security policy stating that confidential data is forbidden from influencing 'public' outputs [5]. This concept was generalized as the *non-interference* property which states that certain classes of computations must not affect others [7].

A wide range of non-interference analyses exist, both static and dynamic ones as well as hybrid combinations. Dynamic analyses are popular because they are more permissive in general, i. e., they reject fewer programs that are in fact safe, and they allow unsafe programs as long as only safe program paths are executed. Further, they can be applied to programming languages such as JavaScript which are not amenable to static analysis and commonly used in settings where code is loaded dynamically.

For example, web pages can include JavaScript code from several different servers. Each of these pieces of code can both read and write the entire document that includes it; this means that confidential personal information known to one server might be exfiltrated to others. Web browsers can use dynamic information flow analysis to track the origins of each piece of data and forbid unwanted flows of possibly sensitive data from one Internet domain to another [11, 9].

Not all applications of information flow analysis are directly related to security or privacy, however; the analysis can also have more general software engineering uses to enforce application-specific properties. For example, an industry partner would like us to verify that their code handling the routing of

*This work was supported by the French National Research Agency (ANR), project AnaStaSec, ANR-14-CE28-0014.

network packets only depends on packet headers but not the payload. In terms of information flow analysis, the packet payload is treated as 'confidential' data that is not allowed to affect the handling of the packet in any way.

In order to be able to enforce such properties, we are developing a hybrid information flow analysis, trying to unify the best features of static and dynamic analyses. Our analysis is aimed at a large subset of the C programming language with the goal of scaling the analysis to real-world safety-critical C applications. The present paper is a step into this direction. We are also developing a machine-checked proof of correctness of the entire approach.

The two main contributions of this paper are thus the following:

- An extension of a previous hybrid information flow analysis for a subset of C that included pointers to scalars; our extension can deal with arrays and pointer arithmetic.
- The formalization of the underlying theory in the Isabelle/HOL proof assistant, a full machine-checked proof of the correctness of our monitor semantics, and ongoing work on formalizing and proving correct our program transformation.

The rest of the paper is organized as follows. Section 2 describes our model of information flow and non-interference. Section 3 describes our fully formalized semantics for information flow monitoring and its formal proof of soundness. Section 4 describes the program transformation implementing the flow monitor for C programs. Section 5 mentions some features of our concrete implementation, Section 6 surveys related work, and Section 7 concludes.

2 Information flow tracking by example

We illustrate the problems of information flow tracking with pointers and arrays in a series of examples. The goal of the information flow analysis is to ensure that all public outputs of the program are independent of all secret inputs. That is, running the program twice with the same public inputs but different secret inputs should give the same public outputs; this property is called *non-interference*. In our case, the analysis dynamically tracks the public/secret status of variables and treats every variable that is public at the end of the program as an output.

In general, there can be more security levels than just public and secret; in that case, they are required to form a finite lattice with the bottom element as the 'most public' security level. We assume an attacker who knows the program's source code and is able to make perfect deductions about secret inputs from observed public inputs and public outputs. We ignore timing, nontermination, and other side channels that may also leak secret information.

For this informal presentation, assume that the variable `secret` is of type `int` and tagged as 'secret'. All other variables are initially public (non-confidential) and of type `int` unless declared otherwise. The dynamic part of the analysis described in this paper works by instrumenting the code with additional monitoring code. Each variable `x` is associated with one or more additional *label variables* marked here by underlining the variable name and adding optional suffixes, e. g., $\underline{x}$. Security levels are tracked as integer values 0 (public) and 1 (secret). Where levels from different sources must be taken into account, they are joined using the `|` (bitwise-or) operator, ensuring that the result is secret iff one of the inputs is secret. Most of the examples in this section follow Assaf's work [1].

Example 1. The code in Figure 1a exhibits *explicit flows* of secret information from `secret` to `x` and then further to `z`; the information flows explicitly via assignments. The analysis must recognize these two variables as secret; their values must not be output, otherwise some information about `secret` would

```
x = secret;
z = x + y;
```

(a) Explicit flow

```
if (secret) {
    x = 0;
} else {
    y = 1;
}
```

(b) Implicit flow

```
int *p;
if (secret) {
    p = &x;
} else {
    p = &y;
}
*p = 1;
```

(c) Pointer-based flow

```
int array[2] = { 0, 0 };
array[secret & 1] = 1;
x = array[0];
```

(d) Array-based flow

Figure 1: Examples of the four kinds of information flow handled by our analysis.

leak. These flows can be monitored by instrumenting the code with the two assignments $\underline{\texttt{x}}$ = $\underline{\texttt{secret}}$; and $\underline{\texttt{z}}$ = $\underline{\texttt{x}}$ | $\underline{\texttt{y}}$; mirroring the original assignments. □

Example 2. Conditional branches cause *implicit flows* from the condition to any assignment controlled by the branch. In Figure 1b, there is an implicit flow from `secret` to `x` and `y`: Inspecting their values may allow an inference whether `secret` is zero or nonzero. Implicit flows are tracked by the control context in a variable $\underline{\texttt{pc}}$ (program counter status), with a new variant $\underline{\texttt{pc}'}$, $\underline{\texttt{pc}''}$, ... for each branching statement. The initial value of the global $\underline{\texttt{pc}}$ is 0 (public), and every branching statement's own $\underline{\texttt{pc}}$ variable is computed as the combination of the directly enclosing $\underline{\texttt{pc}}$ variant and the branch condition's label. The current $\underline{\texttt{pc}}$ variable must be taken into account for any assignment.

Additionally, both branches must update the labels of any variables modified in the *other* branch to ensure that the flow is captured regardless of the actual path taken.

```
pc' = pc | secret;
if (secret) {
    x = 0;
    x = 0 | pc';
    y = y | pc';
} else {
    y = 1;
    y = 0 | pc';
    x = x | pc';
}
```

Note that constants are public and thus get the label 0. As this is the neutral element of the | operator, constants have no influence on the containing expression's status. □

Example 3. An assignment through a pointer introduces a flow from the pointer expression to every possible pointer target. In our monitor, these targets are identified by static points-to analysis and updated with the pointer's label. A label pointer tracks the exact target of the pointer at run time. This means that a pointer variable `p` gets *two* label variables, $\underline{\texttt{p}}$ for the label of the pointer itself and $\underline{\texttt{p\_d1}}$ (of type pointer to label) for the label of `p`'s target. Whenever the program updates `p` to point to a target `t`, the monitor updates $\underline{\texttt{p\_d1}}$ to point to the label $\underline{\texttt{t}}$ of `t`.

The example in Figure 1c is monitored as follows:

```
pc' = pc | secret;
if (secret) {
```

```
    p = &x;
    p = 0 | pc′;
    p_d1 = &x;
} else {
    p = &y;
    p = 0 | pc′;
    p_d1 = &y;
}
*p = 1;
*p_d1 = 0 | pc;
x = x | p | pc;
y = y | p | pc;
```

The updates of x and y are needed to ensure a sound approximation of the respective labels because it is not known statically which of the two variables will actually be overwritten. □

Example 4. The main contributions of this paper concern the handling of arrays. Consider the problem of writing to an array at a secret index as in Figure 1d. The analysis must again treat `x` (and all of `array`) as a secret variable: Outputting `x` or any element of `array` at the end of the program would allow an attacker to infer whether the least significant bit of `secret` is 1. A single secret write to an array element thus taints the entire array. For efficient handling of flows through arrays, we introduce a *summary label* for the entire array which is updated monotonically on each write access to the array.

```
int array[2] = { 0, 0 };
array = 0;
array[secret & 1] = 1;
array |= 0 | secret;
x = array[0];
x = 0 | array;
```

The `|=` operator updates `array` by combining its old value with the right-hand-side value, i. e., it performs the equivalent of `array = array | secret`. Such updates are monotonic, so an array's summary label can never decrease. As we will see later, we often also need field-sensitive tracking of array fields in addition to the summary label. □

Note that all examples except the first share a common property: A piece of code modifies some object, but it is not known statically which one of several objects (variables or array fields) is affected in a concrete execution. The dynamic part of the analysis, i. e., the instrumentation code, must be aware of all possible objects that may be affected and update their statuses to hold a safe over-approximation of the actual status. The set of possible target objects is computed by a standard points-to analysis, the static part of our analysis.

3 Monitor semantics

We can now formalize the intuitive explanations from the previous section as a system of information flow monitoring semantics of programs. The semantics compute a *label memory* Γ which tracks the labels of objects in memory. It is then possible to prove that this semantics satisfies the required non-interference property.

type-synonym *loc* = *block* ∗ *int option*

datatype *val* = *Num int* | *Ptr loc*

datatype *block-val* = *ScalarVal val* | *ArrayVal* (*int* ⇒ *val*)

type-synonym *environment* = *name* ⇒ *block*
type-synonym *memory* = *block* ⇒ *block-val*
type-synonym *label-memory* = *block* ⇒ *label*

Figure 2: Memory model of our simple programming language.

datatype *lval* = *Var name offs* | *Deref expr*
and *expr* = *Const int* | *Lval lval* | *AddrOf lval*
| *BinOp expr expr* (**infixl** ∘ *55*) | *PtrAdd expr expr* (**infix** ⊕ *54*)
and *offs* = *NoOffset* | *Index expr*

Figure 3: Abstract syntax of expressions in our simple programming language. The **infix** annotations in parentheses define syntactic sugar for some operators.

The types, definitions, and proofs described in this section are fully formalized and checked in the Isabelle/HOL proof assistant [15]. Their presentation in the paper was generated automatically, directly from the Isabelle/HOL formalization. As the full development is 1900 lines long, we only show some key parts and omit auxiliary definitions, lemmas, and proofs.[1]

3.1 Expression semantics

We formalize a simple imperative language corresponding to a subset of C. Figure 2 shows the basics of the memory model. The types *name* and *block* are abstract; the type *label* is required to be some bounded lattice with a bottom element $\bot$, a join operation $\sqcup$, and a corresponding partial order $\sqsubseteq$.

A location *loc* is a pair of a *block* and an optional offset. A value *val* is either a number or a pointer to a *loc*. A block can hold a value *block-val*, which is either a scalar *val* or an array of unbounded size represented as a function from *int* to *val*. To simplify the first version of our theory, there are no multi-dimensional arrays: Array elements are scalars of type *val*. An environment E maps names to blocks, a memory M maps blocks to block values, and a label memory Γ maps blocks to their security labels.

Figure 3 shows the abstract syntax of our expressions. The representation is designed to be as close as reasonably possible to the one used by Frama-C [12], which in turn is based on CIL [14]. There is a distinction between lvalue expressions *lval* which evaluate to objects and rvalue expressions *expr* which evaluate to values. An *lval* may be based on a variable or a dereference expression. There is an auxiliary type *offs* for optional offsets into objects, i. e., array indexing. An *expr* may be a constant, the value of an *lval* (obtained using the *Lval* constructor), the address of an *lval*, or a binary operation on *expr*s. We have a generic arithmetic operator $\circ$ that is intended to work on numbers and a pointer addition operator $\oplus$ for adding a pointer and an integer. For simplicity, there are no arithmetic comparisons or boolean operators.

Figure 4 shows the inference rules capturing our definition of the semantics of expressions. The

[1] The entire development is available online at `http://www.complang.tuwien.ac.at/gergo/tini/`.

rules describe both the concrete semantics, i. e., the value computed by an expression, and our monitor semantics, i. e., the security label assigned to the expression. A judgement $E, M, \Gamma \vdash e \rightarrow v, s$ means that in the context of an environment E, a memory M, and a label memory Γ, the expression e evaluates (as an rvalue) to the value v and the security label s. There are corresponding relations $\leftarrow$ for the evaluation of lvalue expressions to locations and $\rightarrow_o$ for offset expressions to optional integers. Offsets *Some i*, which arise from evaluating an *Index* offset expression, are used for array locations only. Scalar locations have offset *None*, which is the value corresponding to a *NoOffset* offset expression.

As an example, consider the C expression `arr[idx]` where `arr` and `idx` are variables. It is represented in the abstract syntax as *Var* `arr` *(Index (Lval (Var* `idx` *NoOffset)))*. For evaluating it as an lvalue, the LVALVAR rule applies, and the memory block for `arr` is determined from the environment E. The index expression `idx` can be evaluated to an integer offset using the OFFSIDX rule and further recursive rule applications of LVALVAR and OFFSNONE.

Note that this presentation does not include static typing of expressions. Using scalar values with an index or array values without an index is a type error, as is interchanging *Num* and *Ptr* values. As usual, the semantics simply gets stuck in such cases. Note also that the generic binary operator $\circ$ is interpreted by some unspecified *eval-binop* function whose details we do not care about.

Expressions' security labels are computed by the semantics by merging the labels of subexpressions using the label lattice's $\sqcup$ operation. Constants and the locations of variables are considered public ($\bot$), while the labels of memory locations are read from the label memory Γ whenever the value of the memory location is read from the memory M in the RVALSCALARLVAL and RVALARRAYLVAL rules.

3.2 Statement semantics

Figure 5 shows the abstract syntax of statements of our target language. The *Skip* statement, program sequencing, *If* and *While* statements are standard. However, for technical reasons (to make proofs tractable), we currently use two different forms of the assignment statement: Plain *Assign* if a value is written to a scalar location and *AssignArrayElem* if a value is written to an array element.

Figure 6 shows the semantics of statements, again describing both concrete semantics (effects of the program on the memory) and monitor semantics (effects on the security label memory). The judgements take the form $E, S_P, pc \vdash program, M, \Gamma \Rightarrow M', \Gamma'$. This means that in the context E, S_P, pc, the program *program* evaluated on a memory M and label memory Γ terminates with a new memory M' and new label memory Γ'. The meaning of the context element S_P will be explained below. E is the environment, and pc is the program counter label.

The concrete parts of the semantics, capturing the computation of the new memory M, are standard. The memory can only be modified by assignment. In the ASSIGNSCALAR rule, the assignment's left-hand side x is evaluated to a location consisting of a memory block b without an offset, i. e., a scalar location. The right-hand side e is evaluated to a value v. The new memory is obtained by updating the value stored at block b in the memory to be *ScalarVal v*. The ASSIGNARRAYELEM rule is similar but more involved. The assignment's left-hand side x evaluates to memory block b with an integer index i. The memory M must contain an array *arr* at block b. The new memory M' is obtained by updating *arr* at position i and storing this new array at block b.

The concrete semantics of *If* statements uses an unspecified function *istrue* of type *val* $\Rightarrow$*bool* to select one of the branches to execute. The concrete semantics of *While* loops evaluates the body once if the condition is true, then re-applies a *While* inference rule in the new memory configuration. A loop terminates iff the condition becomes false at some point, in which case the WHILEF rule applies and performs no further changes to the memory.

$$\frac{E\,x = b \qquad E, M, \Gamma \vdash \mathit{offs} \rightarrow_o \mathit{offset}, s}{E, M, \Gamma \vdash \mathit{Var}\ x\ \mathit{offs} \leftarrow (b, \mathit{offset}), s}\ \textsc{LvalVar}$$

$$\frac{E, M, \Gamma \vdash a \rightarrow \mathit{Ptr}\ (b, \mathit{offs}), s}{E, M, \Gamma \vdash \mathit{Deref}\ a \leftarrow (b, \mathit{offs}), s}\ \textsc{LvalMem}$$

$$\frac{}{E, M, \Gamma \vdash \mathit{Const}\ c \rightarrow \mathit{Num}\ c, \bot}\ \textsc{RvalConst}$$

$$\frac{E, M, \Gamma \vdash a \leftarrow (b, \mathit{None}), sl \qquad M\ b = \mathit{ScalarVal}\ v \qquad \Gamma\ b = sr \qquad sl \sqcup sr = s}{E, M, \Gamma \vdash \mathit{Lval}\ a \rightarrow v, s}\ \textsc{RvalScalarLval}$$

$$\frac{E, M, \Gamma \vdash a \leftarrow (b, \mathit{Some}\ \mathit{idx}), sl \qquad M\ b = \mathit{ArrayVal}\ \mathit{arr} \qquad \mathit{arr}\ \mathit{idx} = v \qquad \Gamma\ b = sr \qquad sl \sqcup sr = s}{E, M, \Gamma \vdash \mathit{Lval}\ a \rightarrow v, s}$$
$$\textsc{RvalArrayLval}$$

$$\frac{E, M, \Gamma \vdash a \leftarrow l, s \qquad p = \mathit{Ptr}\ l}{E, M, \Gamma \vdash \mathit{AddrOf}\ a \rightarrow p, s}\ \textsc{RvalRef}$$

$$\frac{E, M, \Gamma \vdash a \rightarrow \mathit{Num}\ va, sa \qquad E, M, \Gamma \vdash b \rightarrow \mathit{Num}\ vb, sb \qquad \mathit{eval\text{-}binop}\ va\ vb = v \qquad sa \sqcup sb = s}{E, M, \Gamma \vdash a \circ b \rightarrow \mathit{Num}\ v, s}$$
$$\textsc{RvalBinop}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash p \rightarrow \mathit{Ptr}\ (b, \mathit{Some}\ \mathit{idx}), sb \\ E, M, \Gamma \vdash \mathit{offs} \rightarrow \mathit{Num}\ i, si \qquad l = (b, \mathit{Some}\ (\mathit{idx} + i)) \qquad s = sb \sqcup si\end{array}}{E, M, \Gamma \vdash p \oplus \mathit{offs} \rightarrow \mathit{Ptr}\ l, s}\ \textsc{RvalPtrAdd}$$

$$\frac{}{E, M, \Gamma \vdash \mathit{NoOffset} \rightarrow_o \mathit{None}, \bot}\ \textsc{OffsNone} \qquad \frac{E, M, \Gamma \vdash i \rightarrow \mathit{Num}\ \mathit{idx}, s}{E, M, \Gamma \vdash \mathit{Index}\ i \rightarrow_o \mathit{Some}\ \mathit{idx}, s}\ \textsc{OffsIdx}$$

Figure 4: Inference rules defining the semantics of expressions in our example programming language. Judgements compute both concrete values and security label values of expressions.

```
datatype instr = Skip
               | Assign lval expr                (- ::= - 52)
               | AssignArrayElem lval expr       (- ::=' - 52)
               | Seq instr instr                 (infixr ;; 51)
               | If expr instr instr
               | While expr instr
```

Figure 5: Abstract syntax of statements in the example programming language.

$$\frac{}{E, S_P, pc \vdash Skip, M, \Gamma \Rightarrow M, \Gamma}\ \textsc{Skip}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash x \leftarrow (b, None), sl \qquad E, M, \Gamma \vdash e \rightarrow v, sv \qquad s = sl \sqcup sv \sqcup pc \\ s' = sl \sqcup pc \qquad M' = M(b := ScalarVal\ v) \qquad \Gamma' = \Gamma(b := s) \qquad \Gamma'' = update\ S_P\ (x ::= e)\ s'\ \Gamma'\end{array}}{\begin{array}{c}E, S_P, pc \vdash x ::= e, M, \Gamma \Rightarrow M', \Gamma'' \\ \textsc{AssignScalar}\end{array}}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash x \leftarrow (b, Some\ i), sl \qquad E, M, \Gamma \vdash e \rightarrow v, sv \\ s = sl \sqcup sv \sqcup pc \qquad s' = sl \sqcup pc \qquad M\ b = ArrayVal\ arr \qquad M' = M(b := ArrayVal\ (arr(i := v))) \\ \Gamma\ b = l \qquad \Gamma' = \Gamma(b := s \sqcup l) \qquad \Gamma'' = update\ S_P\ (x ::= e)\ s'\ \Gamma'\end{array}}{\begin{array}{c}E, S_P, pc \vdash x ::='\ e, M, \Gamma \Rightarrow M', \Gamma'' \\ \textsc{AssignArrayElem}\end{array}}$$

$$\frac{E, S_P, pc \vdash c_1, M, \Gamma \Rightarrow M', \Gamma' \qquad E, S_P, pc \vdash c_2, M', \Gamma' \Rightarrow M'', \Gamma''}{E, S_P, pc \vdash c_1\ ;;\ c_2, M, \Gamma \Rightarrow M'', \Gamma''}\ \textsc{Comp}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash cond \rightarrow v, s \qquad istrue\ v \\ pc' = s \sqcup pc \qquad E, S_P, pc' \vdash then\text{-}body, M, \Gamma \Rightarrow M', \Gamma' \qquad \Gamma'' = update\ S_P\ else\text{-}body\ pc'\ \Gamma'\end{array}}{E, S_P, pc \vdash If\ cond\ then\text{-}body\ else\text{-}body, M, \Gamma \Rightarrow M', \Gamma''}\ \textsc{IfT}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash a \rightarrow v, s \qquad \neg\ istrue\ v \\ pc' = s \sqcup pc \qquad E, S_P, pc' \vdash else\text{-}body, M, \Gamma \Rightarrow M', \Gamma' \qquad \Gamma'' = update\ S_P\ then\text{-}body\ pc'\ \Gamma'\end{array}}{E, S_P, pc \vdash If\ a\ then\text{-}body\ else\text{-}body, M, \Gamma \Rightarrow M', \Gamma''}\ \textsc{IfF}$$

$$\frac{\begin{array}{c}E, M, \Gamma \vdash cond \rightarrow v, s \qquad istrue\ v \qquad pc' = s \sqcup pc \\ E, S_P, pc' \vdash body, M, \Gamma \Rightarrow M', \Gamma' \qquad E, S_P, pc' \vdash While\ cond\ body, M', \Gamma' \Rightarrow M'', \Gamma''\end{array}}{E, S_P, pc \vdash While\ cond\ body, M, \Gamma \Rightarrow M'', \Gamma''}\ \textsc{WhileT}$$

$$\frac{E, M, \Gamma \vdash cond \rightarrow v, s \qquad \neg\ istrue\ v \qquad pc' = s \sqcup pc \qquad \Gamma' = update\ S_P\ body\ pc'\ \Gamma}{E, S_P, pc \vdash While\ cond\ body, M, \Gamma \Rightarrow M, \Gamma'}\ \textsc{WhileF}$$

Figure 6: Semantics of statements in the example programming language. Both concrete effects on memory and effects on the security label memory are tracked.

fun *collect-updates* :: *alias-function* ⇒ *instr* ⇒ *block set* **where**
collect-updates S_P *Skip* = {} |
collect-updates S_P (x ::= -) = S_P x |
collect-updates S_P (x ::=′ -) = S_P x |
collect-updates S_P (*Seq* i_1 i_2) = *collect-updates* S_P i_1 ∪ *collect-updates* S_P i_2 |
collect-updates S_P (*If* - i_1 i_2) = *collect-updates* S_P i_1 ∪ *collect-updates* S_P i_2 |
collect-updates S_P (*While* - i) = *collect-updates* S_P i

fun *update* :: *alias-function* ⇒ *instr* ⇒ *label* ⇒ *label-memory* ⇒ *label-memory* **where**
update S_P *prog* s Γ = (λb. *if* b ∈ (*collect-updates* S_P *prog*) *then* Γ(b) ⊔ s *else* Γ(b))

Figure 7: Definition of the *update* function used to track the effects of aliasing and unexecuted program paths on the label memory.

The monitor semantics deserves more detailed explanations. Consider first the expression $\Gamma' = \Gamma(b := s)$ in the AssignScalar rule. This updates the security label of the target block b to the new label s, which is computed from the label sl of the assignment's left-hand side's location, the label sv of the right-hand side value, and the current program counter label pc. This captures the direct information flow as shown in Example 1. After this, the label memory is updated again; the final monitor is $\Gamma'' =$ *update* S_P $(x ::= e)$ s' Γ'. This captures pointer-induced flows as demonstrated in Example 3. If the lvalue x is a pointer expression and may refer to different memory locations at runtime, the labels of each corresponding memory block must be updated conservatively. This is done by the *update* function defined in Figure 7. This function takes an alias analysis function S_P, a program fragment, a label s and a label memory Γ. It applies the auxiliary function *collect-updates* to find all memory blocks that may be modified by the given program fragment, then produces a new label memory where the label of every block possibly modified by the program fragment is joined with the label s. In the particular case of the AssignScalar rule, the program fragment passed to *update* is the assignment $x ::= e$ itself, which means that the set of blocks to be updated evaluates to just S_P x. Correctness of the update depends on a correctness criterion for the S_P function itself. Our formalization uses a predicate *admissible* S_P *E M program* (shown in Figure 8) to express that a static analysis S_P computes a safe overapproximation of points-to sets with respect to the given program, environment, and starting memory. An alias function is admissible for a program in a certain configuration if it captures every assignment's target's correctly and is admissible for all possible configurations that arise in the evaluation of subprograms.

The AssignArrayElem rule is similar to AssignScalar in its handling of the label memory. The only difference is in the computation $\Gamma' = \Gamma(b := s \sqcup l)$ where l is the memory block's old security label. This means that an array block's label can only ever increase monotonically, but never decrease. This behavior corresponds to the discussion of Example 4.

The inference rules involving control flow also use the *update* function to capture implicit flows as discussed in Example 2. After executing one of the branches of an *If* statement, *update* is used to adjust the security labels of all the memory blocks that may be modified in the *other* branch. Both the actual execution of one of the branches and the update of the other branch are performed using an updated program counter label pc'. Similarly, even if a *While* loop never iterates, the labels of all the objects that may be modified in its body are updated with pc'. All this ensures that implicit flows are correctly captured: The labels of objects that may be modified under the control of the branch condition are at least as high as the branch condition's label. If the branch condition is secret, these objects become secret as well, and no public information escapes that might allow attackers to infer anything about the condition.

fun *loc-block* :: $loc \Rightarrow block$ **where** $loc\text{-}block\ (b, \text{-}) = b$

fun *admissible* **where**
$admissible\ f\ E\ M\ Skip = True$ |
$admissible\ f\ E\ M\ (x ::= e) = (\forall \Gamma.\forall l.\forall s.\ (E, M, \Gamma \vdash x \leftarrow l, s) \longrightarrow (loc\text{-}block\ l) \in f\ x)$ |
$admissible\ f\ E\ M\ (x ::='\ e) = (\forall \Gamma.\forall l.\forall s.\ (E, M, \Gamma \vdash x \leftarrow l, s) \longrightarrow (loc\text{-}block\ l) \in f\ x)$ |
$admissible\ f\ E\ M\ (Seq\ a\ b) =$
$(admissible\ f\ E\ M\ a \wedge$
$(\forall \Gamma.\forall pc.\forall M'.\forall \Gamma'.\ (E, f, pc \vdash a, M, \Gamma \Rightarrow M', \Gamma') \longrightarrow admissible\ f\ E\ M'\ b))$ |
$admissible\ f\ E\ M\ (If\ c\ t\ e) = (admissible\ f\ E\ M\ t \wedge admissible\ f\ E\ M\ e)$ |
$admissible\ f\ E\ M\ (While\ c\ body) =$
$(admissible\ f\ E\ M\ body \wedge$
$(\forall M.\forall \Gamma.\forall pc.\forall M'.\forall \Gamma'.\ admissible\ f\ E\ M\ body \longrightarrow (E, f, pc \vdash body, M, \Gamma \Rightarrow M', \Gamma') \longrightarrow$
$admissible\ f\ E\ M'\ body))$

Figure 8: Definition of the *admissible* predicate on alias functions.

3.3 Proof of monitor correctness

After describing the monitor semantics, we can now proceed to its proof of correctness. Recall that the goal is to prove non-interference: If a program is run twice on equivalent public inputs but possibly different secret inputs, all the public outputs must be the same on both runs. This ensures that the program's (public) output doesn't allow any inferences about the secret inputs.

The equivalence of public inputs is formalized in the following definition of *s-equivalence*. Two memories M_1 and M_2 are equivalent up to a security label s if they have the same contents for every memory block whose label in a certain security memory Γ is below s:

definition *s-equivalence* :: *label-memory* $\Rightarrow$ *label* $\Rightarrow$ *memory* $\Rightarrow$ *memory* $\Rightarrow$ *bool* (-, - $\vdash$ - $\sim$ -) **where**
$\Gamma, s \vdash M_1 \sim M_2 \equiv (\forall\ b{::}block.\ \Gamma(b) \sqsubseteq s \longrightarrow mem\text{-}equal\ M_1\ M_2\ b)$

(The *mem-equal* predicate captures equality of the values stored in block b in both memories. We omit its definition for brevity.) Somewhat similarly to s-equivalence on memories, we define a predicate imposing a partial ordering on label memories, saying that Γ_2 is *less restrictive than* Γ_1 *up to* s if it respects the $\sqsubseteq$ ordering on all blocks whose labels are below s:

definition *less-restrictive-up-to* :: *label* $\Rightarrow$ *label-memory* $\Rightarrow$ *label-memory* $\Rightarrow$ *bool* (- $\vdash$ - $\sqsubseteq$ -) **where**
$s \vdash \Gamma_2 \sqsubseteq \Gamma_1 \equiv (\forall\ b{::}block.\ \Gamma_1(b) \sqsubseteq s \longrightarrow \Gamma_2(b) \sqsubseteq \Gamma_1(b))$

With these definitions, we can state an important lemma saying that the evaluation of expressions in s-equivalent memories is deterministic in a certain sense:

lemma *expr-evaluation-with-s-equivalence*:
assumes $s \vdash \Gamma_2 \sqsubseteq \Gamma_1$
and $\Gamma_1, s \vdash M_1 \sim M_2$
shows $\forall s_1\ v_2\ s_2.\ s_1 \sqsubseteq s \longrightarrow (E, M_1, \Gamma_1 \vdash a \rightarrow v_1, s_1) \longrightarrow (E, M_2, \Gamma_2 \vdash a \rightarrow v_2, s_2) \longrightarrow v_1 = v_2 \wedge s_2 \sqsubseteq s_1$
and $\forall s_1\ b_2\ s_2.\ s_1 \sqsubseteq s \longrightarrow (E, M_1, \Gamma_1 \vdash b \leftarrow b_1, s_1) \longrightarrow (E, M_2, \Gamma_2 \vdash b \leftarrow b_2, s_2) \longrightarrow b_1 = b_2 \wedge s_2 \sqsubseteq s_1$
and $\forall s_1\ i_2\ s_2.\ s_1 \sqsubseteq s \longrightarrow (E, M_1, \Gamma_1 \vdash c \rightarrow_o i_1, s_1) \longrightarrow (E, M_2, \Gamma_2 \vdash c \rightarrow_o i_2, s_2) \longrightarrow i_1 = i_2 \wedge s_2 \sqsubseteq s_1$

This lemma expresses that if expression evaluation yields a value with a label below s, then evaluating the same expression in an s-equivalent configuration will yield the same value and a smaller (or equal) label. The proof (omitted here) proceeds by mutual induction on the semantics of evaluation of the different kinds of expressions.

Our main result is the formal proof of the following soundness theorem:

theorem *monitor-soundness*:
assumes $E, S_P, pc_1 \vdash program, M_1, \Gamma_1 \Rightarrow M_1', \Gamma_1'$
and *admissible* $S_P\ E\ M_1\ program$
and *admissible* $S_P\ E\ M_2\ program$
and $pc_2 \sqsubseteq pc_1$
and $s \vdash \Gamma_2 \sqsubseteq \Gamma_1$
and $\Gamma_1, s \vdash M_1 \sim M_2$
shows $(E, S_P, pc_2 \vdash program, M_2, \Gamma_2 \Rightarrow M_2', \Gamma_2') \longrightarrow (s \vdash \Gamma_2' \sqsubseteq \Gamma_1') \wedge (\Gamma_1', s \vdash M_1' \sim M_2')$

This theorem shows that running the same program twice in s-equivalent memories M_1 and M_2 (and corresponding side conditions on the program counter labels and security memories) preserves s-equivalence. Inspection of memory blocks whose labels are below s in Γ_1' does not yield any information to an attacker. The result only holds if the static analysis S_P is *admissible* for the given program, i. e., it safely overapproximates all aliasing in the program when started from a given memory configuration.

Proof sketch. The proof of the soundness theorem proceeds by rule induction on the semantics. We will sketch the main idea of the soundness argument for assignments to scalars and for one branch of the evaluation of the *If* statement. In either case, the idea is to show preservation of s-equivalence and the 'less restrictive up to' relation by considering how the value and label of an arbitrary memory block b is modified by the program.

Rule AssignScalar. We may assume that there exist derivations in the semantics showing both $E, S_P, pc_1 \vdash x ::= e, M_1, \Gamma_1 \Rightarrow M_1', \Gamma_1'$ and $E, S_P, pc_2 \vdash x ::= e, M_2, \Gamma_2 \Rightarrow M_2', \Gamma_2'$. In these derivations, name the memory block referenced by x as b_1 and b_2 and the label of evaluating x as an lvalue as s_1 and s_2, respectively. Assume also that there is some arbitrary memory block b where $\Gamma_1'(b) \sqsubseteq s$, i. e., *after* the assignment the label of b is below s. It suffices to show that $\Gamma_2'(b) \sqsubseteq \Gamma_1'(b)$ and $M_1'(b) = M_2'(b)$.

Making a case distinction, assume first that $b \in S_P\ x$. This means that b may be modified by this assignment according to the static analysis. It follows that $s_1 \sqsubseteq s$ since otherwise the *update* function would have changed b's label such that $\Gamma_1'(b) \sqsubseteq s$ would not hold. Using $s_1 \sqsubseteq s$ we can apply the expression evaluation lemma from above to obtain $b_1 = b_2$, i. e., the same block is assigned in both executions. Further, if $b_1 = b$, i. e., this is indeed the block that is modified by the assignment, another application of the lemma ensures that the same value is assigned (showing $M_1'(b) = M_2'(b)$) and that the expression's labels in the two derivation trees respect the $\sqsubseteq$ ordering, establishing $\Gamma_2'(b) \sqsubseteq \Gamma_1'(b)$. Otherwise, if $b_1 \neq b$, then the memory at b is not modified at all, and its label is updated safely using *update*, again establishing the intended results.

Finally, in the other case $b \notin S_P\ x$. Because S_P is an admissible analysis, it follows that b is not modified by this assignment. Hence the semantics rule modifies neither the memory nor the label memory, and the result follows directly from the assumptions.

Rule IfT. Assume there is a derivation showing $E, S_P, pc_1 \vdash$ *If c then-body else-body*$, M_1, \Gamma_1 \Rightarrow M_1', \Gamma_1''$ where the condition c evaluates to a true value with label s_1. From the evaluation of the true branch *then-body* in the starting context with updated program counter $pc_1' = s_1 \sqcup pc_1$ obtain a label memory Γ_1' where $\Gamma_1'' =$ *update* S_P *else-body* $pc_1'\ \Gamma_1'$. Assume further there is a derivation showing $E, S_P, pc_2 \vdash$ *If c then-body else-body*$, M_2, \Gamma_2 \Rightarrow M_2', \Gamma_2''$. Note that we do not assume that this derivation enters the same branch. Fix again a block b with $\Gamma_1''(b) \sqsubseteq s$.

Making a case distinction, assume $s_1 \sqsubseteq s$. Using the expression evaluation lemma, the branch condition c evaluates to a true value in the second configuration as well, so the same branch is executed. The required result follows by induction on the execution of *then-body*.

Otherwise, $s_1 \not\sqsubseteq s$. The two derivations may execute different branches; we show that the block b is not affected by the *If* statement at all, so the different executions make no difference to its value or

label. First, we have $b \notin$ *collect-updates* S_P *else-body* because otherwise the *update* function on *else-body* would have raised its label such that the assumption $\Gamma_1''(b) \sqsubseteq s$ could not hold. Otherwise, if $b \in$ *collect-updates* S_P *then-body* were to hold, then at some point during the execution of the *If* statement its label would have to be raised to at least s_1, again violating the assumption. Thus we obtain $\Gamma_2''(b) \sqsubseteq \Gamma_1''(b)$ and $M_1'(b) = M_2'(b)$.

The rules for the other branch of the *If* and for the *While* statement follow similar reasoning. Finally, the proof for evaluation of *Skip* is trivial, and the proof for program composition follows directly from the induction hypothesis for the subprograms. □

The full, completely machine-checked Isabelle/HOL proof of this theorem is about 600 lines long, plus about 200 lines of proofs of key auxiliary lemmas. The structure of the proof itself follows the work of Assaf [1], which gives a manually typeset paper proof of a little more than five pages (without handling arrays). We were able to reproduce the paper proof mostly faithfully, repairing some typographical errors and minor glitches along the way. The most important issue was that Assaf's proof of the assignment rule is too weak: His proof only shows $\Gamma_2'(b) \sqsubseteq s$ (for a block b modified by the assignment) rather than the stronger result $\Gamma_2'(b) \sqsubseteq \Gamma_1'(b) \sqsubseteq s$ needed to establish the goal $s \vdash \Gamma_2' \sqsubseteq \Gamma_1'$. However, it was easy to reuse the structure of the given proof and strengthen it to prove the necessary condition.

4 Program transformation

Given the abstract semantics from the previous section, we now turn to the question of how to implement the security monitor in practice. We want to insert monitoring code into a given program that tracks security labels. At the end of the execution of the program, the label variable $\underline{\texttt{x}}$ for each original program variable `x` should have the same value as $\Gamma(E(\texttt{x}))$ in the monitor semantics. The soundness proof of the monitor then carries over to the analysis code.

4.1 Information flow monitoring without pointers

Without pointers or arrays, inlining the dynamic analysis code is simple: Whenever a variable `x` is read or written, we insert appropriate reads or writes of the corresponding label variable $\underline{\texttt{x}}$. Additionally, for every statement affecting control flow, a new program counter status variable is created and updated as in the monitor semantics in Figure 6. Additional assignments are inserted to model the effects of the control flow branch not taken, as with the *update* function in the monitor semantics.

The difficulties arise when pointers are used: What is the label variable corresponding to a pointer dereference expression `*p`? In the abstract theory, such expressions evaluate to memory blocks b which are used to access both the memory M and the label memory Γ. However, these memory blocks are not available as first-class objects in C, so we need a different way of finding the correct label variable to access.

4.2 Information flow monitoring with pointers to scalars

The solution for tracking pointers developed by Assaf [1], which we follow, is to mirror all pointer structures in the original program in the information flow monitor. For this purpose, each pointer `p` of type $T*^{(n)}$ (i. e., that may be dereferenced n times) is associated with n label pointers $\underline{\texttt{p\_d1}}$, …, $\underline{\texttt{p\_d}n}$. The intention is to ensure that at any point in the program, the expression $*^{(i)}\underline{\texttt{p\_d}i}$ for all $1 \leq i \leq n$ evaluates to the label of $*^{(n)}\texttt{p}$.

For example, if pointer `p` is made to point to variable `x` by an assignment `p = &x` in the original program, a corresponding label pointer variable `p_d1` is made to point to the label variable `x` by the inserted assignment `p_d1 = &x`. Reads and writes through `*p` can then be mirrored in the analysis as reads and writes through `*p_d1`. Assaf gives a formal definition of this transformation and proves that it preserves the invariant that for all pointers in the program, a pointer `p` points to a target `x` iff the corresponding label pointer points to the target's label. This allows a proof of the correctness of the transformation, i. e., it establishes that the instrumented program computes the same security labels as the label memory Γ in the underlying semantics.

4.3 Information flow monitoring with arrays

We extended the approach described above to handle arrays. Note that the monitor semantics in Figures 4 and 6 assume that the memory block storing an array has a single security label, not individual labels for individual array elements. The reason for this was touched on in Example 4: If an array element is written at a secret index, reading another array element and finding it has a non-secret label would leak information about the value of the index.

For this reason, we associate each array `a` with a single label variable `a` called the *summary label*. As in the AssignArrayElem inference rule in the semantics, every write to an array element triggers a *weak update* of this label: The summary label l is not overwritten by the new label s (which incorporates the labels of the index and the value to be written) but with the joined value $s \sqcup l$. As security labels form a lattice, we have $s \sqsubseteq s \sqcup l$ and $l \sqsubseteq s \sqcup l$. This means that over a sequence of assignments to elements of the array with labels $s_1, \ldots, s_n$, the values of the summary label $l_1, \ldots, l_n$ always form an ascending chain with respect to $\sqsubseteq$. Furthermore, at any point, the current l_i is a safe overapproximation of all $s_1, \ldots, s_{i-1}$ written so far. Our analysis ensures that the label of any read from array `a` incorporates its summary label `a`. This means that, if at any point in the program a secret value or secret index is used in an assignment to an element of `a`, all future reads will be treated as secret. This property ensures the equivalence of `a` to the label $\Gamma(E(\texttt{a}))$ and hence the soundness of our information flow analysis in this aspect.

The summary field also plays an important role in handling pointers to array elements as well as pointer arithmetic. Consider the following program fragment:

```
p = &a[i];
p++;
*p = 42;
```

This code assigns the address of array element `a[i]` to pointer `p`, increments `p` to point to the next array element, then writes to memory through `p`. This final write affects an element of the array `a`, so we must ensure that our analysis updates the summary label `a` correctly.

To this end we must ensure that a label pointer associated with `p` always points to the target's summary label and is not moved by pointer arithmetic. In the example above, a summary pointer `p_summary` must be generated by the analysis and pointed to the address of `a`. This pointer is not affected by indexing or pointer arithmetic, i. e., it always points to `a` regardless of the value of the index expression `i` and regardless of the pointer increment using `++`. The assignment through `*p` can then be mirrored in the analysis by a weak update through `*p_summary`, which results in a weak update of `a` as required.

In the presence of arrays of pointers, a summary label is not enough, however: We must additionally track pointer relationships in an array-field-sensitive way. Consider a slightly modified version of the example above, where `a` is now an array of pointers rather than an array of numbers as before, and `p` is therefore a pointer to a pointer:

datatype *type* = *TInt* | *TPtr type* | *TArray type nat*
datatype *label-kind* = *Exact* | *Summary*
datatype *label-type* = *Label label-kind nat type*

fun *ptr-label* **where** *ptr-label* (*Label kind d t*) = *Label kind* (*d+1*) (*TPtr t*)
fun *array-label* **where** *array-label len* (*Label kind d t*) = *Label kind d* (*TArray t len*)

fun *labels-aux* **where**
labels-aux TInt = [*Label Exact 0 TInt*] |
labels-aux (*TPtr t*) =
[*Label Exact 0 TInt*, *Label Summary 1* (*TPtr TInt*)] @ *map ptr-label* (*labels-aux t*) |
labels-aux (*TArray t len*) = *map* (*array-label len*) (*labels-aux t*)

fun *labels* **where**
labels (*TArray t len*) = [*Label Summary 0 TInt*] @ *labels-aux* (*TArray t len*) |
labels t = *labels-aux t*

Figure 9: Computation of label types in the presence of arrays and pointers.

```
p = &a[i];
p++;
a[i+1] = &x;
**p = y;
```

Here the final assignment through `**p` is an assignment to the variable `x`, and the dynamic information flow analysis must therefore be able to execute an appropriate update of its label `x`. Thus there must be an appropriate label pointer `p_d2` where `**p_d2` is the object `x`.

We achieve this by associating a second label with each array of pointers `a[`n`]`: Besides the scalar summary label `a`, we also use an array of label pointers `a_d1[`n`]`. The intention is to ensure that if `a[`i`]` points to `x`, then `a_d1[`i`]` points to `x`. In the example above, we can let `**p_d2` point to `a_d1[i]` initially and then mirror the pointer arithmetic `p++`. We arrive at the following fragment of monitoring code (ignoring summary labels for simplicity):

```
p_d2 = &a_d1[i];
p_d2++;
a_d1[i+1] = &x;
**p_d2 = y | p;
```

The generated code ensures that at the last assignment, `p_d2` points to `a_d1[i+1]`, which in turn points to `x`. The last assignment thus updates `x` as required.

We can thus summarize the requirements for our analysis: Every array `a` needs a summary label `a` and an array of exact labels `a_d0`. Every pointer `p` needs a label `p` for the pointer itself as well as a summary label pointer `p_d1_summary` to point to `p`'s target's summary label and a label pointer `p_d1` to point to `p`'s exact target's label. These rules must be applied recursively for types of nested pointers or arrays, adjusting the number of possible dereferences (`d`). Figure 9 shows how we compute the list of types and dereferencing levels using the function *labels*. The recursive computation is captured in the function *labels-aux*. The most subtle issue is that *labels* must add an outermost summary label for array types.

For a C type declaration `int *b[10]`, encoded as *TArray (TPtr TInt) 10*, this system computes the following label types, which our program transformation turns into the appropriate type declarations:

```
p = &a[i];
*p = 42;
p += secret;
*p = 43;
```

```
p = & a[i];
p = 0 | (i | pc);
p_d1_summary = & a;
p_d1 = & a_d0[i];
*p = 42;
*p_d1_summary |= 0 | (p | pc);
*p_d1 = 0 | (p | pc);
p += secret;
p |= secret | pc;
p_d1_summary = p_d1_summary;
p_d1 += secret;
*p = 43;
*p_d1_summary |= 0 | (p | pc);
*p_d1 = 0 | (p | pc);
```

Figure 10: Example of dynamic information flow monitoring with arrays and pointer arithmetic. The original program (left) is turned into the program with inlined analysis code (right). Our transformation tool's output was modified to make status variable names more readable, changing names like `p_status` to `p`.

```
[Label Summary 0 TInt,                    int b_status;
 Label Exact 0 (TArray TInt 10),          int b_status_d0[10];
 Label Summary 1 (TArray (TPtr TInt) 10), int *b_status_d1_summary[10];
 Label Exact 1 (TArray (TPtr TInt) 10)]   int *b_status_d1[10];
```

Putting everything together, Figure 10 shows another variant of the examples above and the complete dynamic information flow monitoring code generated by our system. In the statement performing pointer arithmetic, we use a variable `secret` to make the flow more visible: When the pointer `p` has been offset by `secret`, its label is joined with `secret`'s label. At the subsequent assignment to `*p`, this label is propagated to the target's label. Observe also how pointer expressions for summary labels perform weak updates (using the `|=` operator), but the corresponding exact labels receive strong updates.

The remaining challenge is to complete the formalization of this program transformation in Isabelle/HOL. The key is a precise statement of the invariant that whenever a pointer expression `p` points to a variable `x`, the corresponding label pointer expression `p` points to `x`. We will then show that the assignments inserted by the program transformation preserve this invariant, which will allow us to establish a complete soundness proof.

5 Implementation notes

We have implemented the program transformation sketched above as a plugin in the modular C analysis and transformation framework Frama-C [12]. The current prototype handles programs with arrays and pointers. For the alias analysis S_P needed by the transformation, we rely on Frama-C's built-in Value analysis, which computes both aliases for pointers and value approximations for numeric variables using intervals and other domains. The transformation is implemented as a transformation of the Frama-C AST, which can then be output as C code. At the time of writing, some details of real-world C programs are not yet handled by the analysis, which precludes us from giving a detailed experimental evaluation.

Using Frama-C's support for code annotations, we allow security levels of variables to be specified as `/*@ public */` or `/*@ private */` at the point of declaration. The corresponding label variables are then initialized accordingly. Labels are tracked as integer values of 0 (public) and 1 (private) and are efficiently combined using the bitwise-or operator `|`. We do not currently support more general lattices; however, extending the current implementation to lattices that can be represented as bitvectors (up to 64 bits) is straightforward.

Users may also insert annotations like `/*@ assert security_status(x) == public; */` in their programs. Such annotations may also occur as function preconditions using Frama-C's annotation language ACSL; for example, any output function could require its arguments to be public. This allows users full freedom to specify their application-specific information flow policies. For example, functions that may cause information to be written to network sockets (such as the common `send(1)` system call) may have contracts requiring their inputs to be public. As another example, cryptographic code may be annotated to ensure that branch conditions are always independent of the cryptographic keys; otherwise, key-dependent control flow may cause differences in timing or other side-channels observable by attackers [3, 6]. Without such annotations, our analysis never reports a policy violation, i. e., without a user-defined policy everything is permitted. As such policies are inherently application-specific, we want to keep our analysis as general as possible and do not specialize it for particular flow policies.

Transformed, annotated programs often contain enough information for the Value analysis to be able to prove such assertions without having to execute the instrumented program at all. Thus our hybrid analysis combined with the powerful components of the Frama-C framework can often be used as a powerful static analysis as well.

6 Related work

As mentioned several times throughout the paper, our work is heavily based on the formulation of information flow monitoring by Assaf et al. [2, 1]. This work only handles pointers to scalars; we have formalized this theory in Isabelle/HOL, extended it to handle arrays, and are working on extending it further. Our concrete implementation of the analysis in Frama-C is also based on the prototype developed by Assaf.

Besides this prototype, we are aware of two implementations of dynamic information flow analysis that aspire to handle real-world programs. Both of these are designed for JavaScript and intended for settings with dynamic code loading. In contrast, our approach assumes a complete program in a C-like language on which a static points-to analysis can be run. The approach by Kerschbaumer et al. [11] handles arrays, but the details are not described; the authors only mention that an array may 'consist[...] of heterogeneously labeled fields'. This heterogeneous labeling is something our approach consciously avoids for soundness reasons, to avoid information leaks through array indices. In our approach, reading an array element always involves reading the array's summary label (see Example 4). The authors do not describe any formal or informal proof of non-interference for their analysis.

The other well-developed analysis for JavaScript is JSFlow [9] with its extended hybrid version [8]. Both track the labels of array elements precisely, but a different notion of non-interference from ours is used: In this variant, it is not allowed to assign secret values to locations that previously held public values (the converse, overwriting a secret value by a public value, is allowed). The monitor aborts the program if a violation of this policy is detected. In our approach, this would correspond to adding an assertion to every assignment statement. In contrast, our approach is more permissive and only uses such constraints at user-defined program points; as discussed above, our analysis is completely independent

of any specific flow policy. The authors prove non-interference of both versions of JSFlow.

In the literature, there are various static information flow analyses, often formulated as flow-sensitive type systems [17, 10], as well as further hybrid static/dynamic analyses somewhat comparable to ours [13, 16]. Arrays are occasionally mentioned in connection with type systems [17] but, to our knowledge, never for the systems involving some dynamic monitoring. As our work shows, arrays raise subtle soundness issues, in particular when combined with pointers and pointer arithmetic; to our knowledge, we are the first ones to handle these issues in detail for a C-like language.

The terminology of weak and strong updates is borrowed from pointer analysis [4].

7 Conclusions and future work

We presented a hybrid information flow analysis for the C programming language with pointers, arrays, and pointer arithmetic. Our analysis is implemented by instrumentation code that tracks information flows by managing security labels associated with each object in the program. As in previous work, pointers to labels mirror pointers to data in the original program. We extend this to arrays, tracking flows both in a field-sensitive way and as a safe overapproximation in a separate summary field for each array. Our analysis is implemented using the Frama-C program analysis and transformation framework.

A machine-checked proof of the correctness of the monitor semantics was formalized using Isabelle/HOL. We will also formalize the program transformation and prove its correctness; adding the required static typing support to our dynamically typed semantics is ongoing work.

We will further extend this work to handle structures in a field-sensitive way. We also intend to use pointer analysis information to allow us to handle type casts between pointer types.

References

[1] Mounir Assaf (2015): *From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow*. Ph.D. thesis, Université de Rennes 1. Available at `https://hal.inria.fr/tel-01184857`.

[2] Mounir Assaf, Julien Signoles, Frédéric Tronel & Éric Totel (2013): *Program Transformation for Non-interference Verification on Programs with Pointers*. In Lech J. Janczewski, Henry B. Wolfe & Sujeet Shenoi, editors: *Security and Privacy Protection in Information Processing Systems*, *IFIP Advances in Information and Communication Technology* 405, Springer, pp. 231–244. Available at `http://dx.doi.org/10.1007/978-3-642-39218-4_18`.

[3] David Brumley & Dan Boneh (2003): *Remote Timing Attacks are Practical*. In: *Proceedings of the 12th Usenix Security Symposium*. Available at `https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf`.

[4] David R. Chase, Mark Wegman & F. Kenneth Zadeck (1990): *Analysis of Pointers and Structures*. *SIGPLAN Not.* 25(6), pp. 296–310, doi:10.1145/93548.93585. Available at `http://doi.acm.org/10.1145/93548.93585`.

[5] Dorothy E. Denning & Peter J. Denning (1977): *Certification of Programs for Secure Information Flow*. *Commun. ACM* 20(7), pp. 504–513, doi:10.1145/359636.359712. Available at `http://doi.acm.org/10.1145/359636.359712`.

[6] Daniel Genkin, Lev Packmanov, Itamar Pipman & Eran Tromer (2016): *ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs*. In: *RSA Conference Cryptographers' Track (CT-RSA)*, *LNCS* 9610, pp. 219–235. Available at `https://eprint.iacr.org/2016/129.pdf`.

[7] J.A. Goguen & J. Meseguer (1982): *Security Policies and Security Models*. In: *Security and Privacy, 1982 IEEE Symposium on*, pp. 11–11, doi:10.1109/SP.1982.10014.

[8] D. Hedin, L. Bello & A. Sabelfeld (2015): *Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language*. In: *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pp. 351–365, doi:10.1109/CSF.2015.31.

[9] Daniel Hedin, Arnar Birgisson, Luciano Bello & Andrei Sabelfeld (2014): *JSFlow: Tracking Information Flow in JavaScript and Its APIs*. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, ACM, pp. 1663–1671, doi:10.1145/2554850.2554909. Available at `http://doi.acm.org/10.1145/2554850.2554909`.

[10] Sebastian Hunt & David Sands (2006): *On Flow-sensitive Security Types*. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, ACM, pp. 79–90, doi:10.1145/1111037.1111045. Available at `http://doi.acm.org/10.1145/1111037.1111045`.

[11] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler & Michael Franz (2013): *Information Flow Tracking Meets Just-in-time Compilation*. *ACM Trans. Archit. Code Optim.* 10(4), pp. 38:1–38:25, doi:10.1145/2555289.2555295. Available at `http://doi.acm.org/10.1145/2555289.2555295`.

[12] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2015): *Frama-C: A software analysis perspective*. *Formal Aspects of Computing* 27(3), pp. 573–609, doi:10.1007/s00165-014-0326-7. Available at `http://dx.doi.org/10.1007/s00165-014-0326-7`.

[13] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen & David A. Schmidt (2007): *Automata-based Confidentiality Monitoring*. In: *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, Springer-Verlag, pp. 75–89. Available at `http://dl.acm.org/citation.cfm?id=1782734.1782741`.

[14] George C. Necula, Scott McPeak, Shree Prakash Rahul & Westley Weimer (2002): *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. In: *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, Springer-Verlag, pp. 213–228. Available at `http://dl.acm.org/citation.cfm?id=647478.727796`.

[15] Tobias Nipkow, Markus Wenzel & Lawrence C. Paulson (2002): *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag.

[16] A. Russo & A. Sabelfeld (2010): *Dynamic vs. Static Flow-Sensitive Security Analysis*. In: *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pp. 186–199, doi:10.1109/CSF.2010.20.

[17] Dennis Volpano, Cynthia Irvine & Geoffrey Smith (1996): *A Sound Type System for Secure Flow Analysis*. *J. Comput. Secur.* 4(2-3), pp. 167–187. Available at `http://dl.acm.org/citation.cfm?id=353629.353648`.