A Dafny-based approach to thread-local information flow analysis

Graeme Smith
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia
ORCID 0000-0003-1019-4761

***Abstract*—The Dafny program verifier supports proofs of functional correctness of single-threaded programs written in an imperative, object-based or functional style. In this paper, we show how Dafny can also be used to support proofs of information flow security in multi-threaded programs. For generality, information flow is analysed with respect to a user-defined lattice of security values, and the security classifications of program variables are value-dependent, i.e., they are not fixed but depend on the current program state. For scalability, our multi-threaded analysis is carried out thread locally using rely/guarantee reasoning. The required well formedness properties of our security lattices and rely and guarantee conditions are proven using Dafny lemmas.**

***Index Terms*—information flow, concurrency, program verifiers, rely/guarantee reasoning, Dafny**

I. Introduction

Information flow analyses track the flow of data through a program, and can hence detect when sensitive data flows to program locations which are considered to be accessible by an attacker. Such analyses range from simple security type systems [1], [2] to more advanced logics which use predicates to represent *value-dependent* security classifications of program variables, i.e., security classifications which evolve as the program executes [3], [4], [5]. These latter approaches have reached a high level of maturity over the last decade in terms of the systems to which they can be applied. In particular, Murray et al. [3], [4] and Ernst and Murray [6] have developed the first practical information flow logics supporting value-dependent security classifications for concurrent programs. Building on these results, information flow logics capable of handling arbitrarily complex interactions between threads using rely/guarantee reasoning [7], [8] have been developed by Coughlin and Smith [9] and Winter et al. [5].

As the complexity of these logics increases, so does the complexity of associated tool support. Much of the work on more expressive logics is supported only by interactive theorem proving [3], [4], [9]. Winter et al. [5] take this further by automating their proofs in Isabelle/HOL [10]. While this allows for experimentation with their logic and provides strong assurance due to the use of verified rules, it is only fully automated for small examples with simple types for which there is a large amount of existing support in Isabelle/HOL.

Ernst and Murray [6] take an alternative approach building a custom symbolic execution tool for their logic in Scala and employing Z3 [11] as a backend for discharging proof obligations. While the logic encoded in their tool is proven to be sound in Isabelle/HOL, proving that the tool's implementation conforms to the logic is significantly harder.

In this paper, we look at the possibility of using an existing program verification tool, one that has been developed for verifying functional correctness of single-threaded programs, for value-dependent information flow analysis of concurrent programs. Such tools can be considered more trustworthy simply because they have larger user bases (and hence many bugs will have been found). This is particularly the case when they share a common back-end verifier, such as Boogie [12] or Why3 [13], with many other tools. Furthermore, there have been efforts to validate these back-ends, such as the approach by Pathasarathy et al. [14] in which Boogie's reasoning steps are translated to a form which can be checked in Isabelle/HOL.

Specifically, we investigate the use of the Boogie-based verifier for Dafny [15] although our approach could be applied to similar tools for more widely used languages such as Frama-C [16] (for C), VerCors [17] (for C and Java) or Prusti [18] or Creusot [19] (for Rust). The paper provides a proof-of-concept that such tools can be used in the context of thread-local information flow analysis. Where we use Dafny-specific notation, we suggest alternative encodings that could be used in other tools.

For information flow analysis, the basic idea is to extend a program we wish to analyse with additional variables whose type belongs to a user-defined security lattice, and whose values correspond to (i) the user-defined security classifications of the original program variables, and (ii) the security levels of information held by them. Assertions employing lattice operators to combine and compare the additional variables can then be added to the program's code to capture standard information flow checks.

For concurrency, the idea is to use rely/guarantee reasoning to focus on a single thread at a time. We model each thread as a method with:

(i) A method call before and after each line of code corresponding to the other threads in its environment taking one or more steps. Analysis of the thread can *rely* on the threads in its environment only behaving in ways consistent with the called method's specification.

(ii) An assertion after each line of code checking that the line of code *guarantees* behaviour consistent with the "rely" methods of other threads in the environment. This compatibility between threads is checked using Dafny lemmas.

Like the security lattice and security classifications, the user needs to provide the rely and guarantee conditions of each thread, as well as any loop invariants that are needed for Dafny to reason about the code. Given these specifications, our encoding of required verification conditions is purely based on the program syntax and hence can be readily automated using a simple front-end transpiler.

We begin in Section II with an overview of the Dafny language focusing on those aspects relevant to our approach. In Section III we discuss value-dependent information-flow analysis and how it can be encoded in Dafny. Similarly, we show how rely/guarantee reasoning can be encoded in Dafny in Section IV. In Sections V and VI, we consider programs with dynamic thread creation and (potentially recursive) method calls, respectively. We show both how information flow logics can be extended to handle them, and how they can be encoded in Dafny. A producer-consumer case study is provided in Section VII. We conclude with a brief discussion of implementing a tool based on our approach in Section VIII.

II. Dafny

The Dafny programming language [15] supports functional, imperative and object-based programming (supporting *traits*, similar to Java interfaces, but not full inheritance). In this paper, we will focus on the core imperative programming constructs of Dafny (assignments, if statements and while loops), and use classes as a means to set up a context in which variables can be shared.

A simple imperative Dafny program is shown in Figure 1a. The method Mult recursively computes the product of two non-negative integer inputs, a and b, and returns it in an integer output x (the keyword var in the else branch is used to introduce the local variable y). The assert clause is checked statically by Dafny's verifier, and does not appear in the compiled code.

The pre- and postcondition of the method are captured in the requires and ensures clauses, respectively. These are used by the verifier to statically verify the method's code. The postcondition makes use of the function mult which returns the product of its inputs, and is proved to be commutative using the lemma Comm; the lemma has an empty body since Dafny can prove it automatically, otherwise the body of a lemma needs a proof provided by the user [20].

Functions which return a Boolean value can be written using the predicate keyword. For example, the predicate in Figure 1b returns true when inputs a and b are non-negative.

Within a Dafny class, we can also have functions, predicates and lemmas that refer to two states, the current state and an old state of the class. For example, consider the class C in Figure 1c in which the two-state predicate P specifies that the value of the state variable x in the current state is greater than that in the old state. Note that the predicate has a *read frame* stating that it can read this, i.e., the state variables of the class. Similarly, the method M has a *write frame* stating that it can modify this. The frame of a function also needs to include any arrays it reads, and that of a method any arrays it writes to. The assertion in which the predicate is used refers to the old state as being that at label L, i.e., the old state is the state before the second line of the code is executed. In the absence of a label for the old state, the old state will default to the state at the start of the method.

```
function mult(a:int, b:int):int {
  a*b
}

lemma Comm(a:int, b:int)
  ensures mult(a, b) == mult(b, a)
{}

method Mult(a:int, b:int) returns (x:int)
  requires a >= 0 && b >= 0
  ensures x == mult(a, b)
{
  if a == 0 {
    x := 0;
  } else {
    var y := Mult(a – 1, b);
    assert y == a*b – b;
    x := y + b;
  }
}
```

(a) Imperative Dafny program comprising a recursive method with an assertion, a function and a lemma.

```
predicate NonNeg(a:int, b:int) {
  a >= 0 && b >= 0
}
```

(b) Dafny predicate.

```
class C {
  var x:int

  twostate predicate P()
    reads this
  {
    x > old(x)
  }

  method M(y:int)
    modifies this
    ensures x == y + 1
  {
    x := y;
    label L: x := x + 1;
    assert P@L();
  }
}
```

(c) Dafny class with a two-state predicate and a label.

Fig. 1: Simple examples of Dafny usage.

III. Information flow

In an information flow analysis [2], program variables are given a security classification from a lattice L of security levels. The lattice is often a simple Boolean lattice with

one level *high*, representing classified information, and one level *low*, representing information that is publicly available. However, it can be arbitrarily complex with different levels representing, for example, the different information access rights within an organisation.

A general lattice can be encoded in Dafny as an enumerated type along with a predicate *leq* defining when a level $l1$ is less than or equal to a level $l2$ (i.e., $l_1 \sqsubseteq l_2$), and functions for returning the *meet* (i.e., greatest lower bound $\sqcap$) and *join* (i.e., least upper bound $\sqcup$) of any two levels. For example, a standard diamond lattice where level A is higher than levels B and C which are in turn higher than level D (but B and C are not ordered) can be encoded as follows.

```
datatype L = A | B | C | D

predicate leq(l1:L, l2:L) { l1 == D || l1 == l2 || l2 == A }

function join(l1:L, l2:L):L {
  if leq(l1, l2) then l2 else if leq(l2, l1) then l1 else A
}

function meet(l1:L, l2:L):L {
  if leq(l1, l2) then l1 else if leq(l2, l1) then l2 else D
}
```

Since the lattice is specific to the program being analysed, this encoding would need to be provided by the user. To ensure it is indeed a lattice, lemmas to prove that leq is a partial order, and that join and meet are the least upper bound and greatest lower bound, respectively, are added. These lemmas are not program-specific and hence can be automatically added to the Dafny file.

```
lemma partialorder(l1:L, l2:L, l3:L)
  ensures leq(l1, l1)
  ensures leq(l1, l2) && leq(l2, l1) ==> l1 == l2
  ensures leq(l1, l2) && leq(l2, l3) ==> leq(l1, l3)
{}

lemma joinLemma(l1:L,l2:L,l3:L)
  ensures leq(l1, join(l1, l2)) && leq(l2, join(l1, l2))
  ensures leq(l1, l3) && leq(l2, l3) ==> leq(join(l1, l2), l3)
{}

lemma meetLemma(l1:L, l2:L, l3:L)
  ensures leq(meet(l1, l2), l1) && leq(meet(l1, l2), l2)
  ensures leq(l3, l1) && leq(l3, l2) ==> leq(l3, meet(l1, l2))
{}
```

Information flow analyses usually operate on one line of code at a time (in either a forwards or backwards direction) utilising standard lattice operators on L to prove *noninterference* [21]. For a Boolean lattice, noninterference amounts to *high* information not affecting the values of *low* variables. This prevents an attacker who can observe *low* variables deducing anything about the *high* values. For a general lattice, noninterference amounts to no variable being influenced by a value at a security level higher than its security classification.

In most information flow analyses, the security classification of program variables is fixed. In more general *value-dependent* approaches [3], [4], [5], they are allowed to change as the program's state evolves. The security classification of a variable x, denoted $\mathcal{L}(x)$, is a state-dependent expression which evaluates to a security level. For the lattice above, for example, we could define a variable x to have security classification B when $z = 0$ and D otherwise. In Dafny, this can be encoded (by the user) as a function within a class which would also include the program to be analysed (as a method).

$$\textsc{Assign}\ \frac{\Gamma_E(e) \sqsubseteq \mathcal{L}(x) \qquad x \in \mathcal{C} \Rightarrow \mathit{secure\_update}(x,e)}{\Gamma, P \ \ \{x := e\} \ \ \Gamma[x \mapsto \Gamma_E(e)], \mathit{sp}(x := e, P)}$$

where $\mathit{secure\_update}(x,e) \mathrel{\widehat{=}} \forall\, y \in \mathit{ctrled}(x) \cdot \Gamma_E(y) \sqsubseteq \mathcal{L}(y)[x \backslash e]$.

Fig. 2: Forwards assignment rule (based on [4]).

```
class C {
  var x:int
  var z:int

  function L_x():L
    reads this
  {if z == 0 then B else D}

  ... // rest of class including the program to be analysed
}
```

The program variables whose values affect $\mathcal{L}(x)$, e.g., z above, are referred to as *control variables*. We let $\mathcal{C}$ denote the set of control variables of a program and $\mathit{ctrled}(z)$ be the set of variables controlled by $z \in \mathcal{C}$.

A. Assignment statements

A typical rule for assignment when working forwards through the code is given in Figure 2. To enable security violations to be detected, the logic keeps track of the current program context in Γ, which maps variables to the security level of the data they hold, and P, a predicate on program variables describing the current state.

The rule has two premisses. The first checks that the security level of the expression e, denoted $\Gamma_E(e)$, is not greater than the security classification of the updated variable x (thus ensuring noninterference). The second checks that if x is a control variable then the security classification of any controlled variables does not fall below the security level of the information they hold. Note that $\mathcal{L}(y)[x \backslash e]$ denotes $\mathcal{L}(y)$ with all occurrences of x replaced by e. Hence, it denotes the value of $\mathcal{L}(y)$ after the assignment $x := e$ has occurred.

If these premisses hold, the assignment is secure and the rule updates the context accordingly (*sp* is strongest postcondition). If either of the premisses fail, the analysis flags a security violation. A similar rule for the backwards direction is given in Figure 3 (where *wp* is weakest precondition and $x, y := e, f$ denotes simultaneous assignment). The rule introduces two proof obligations identical to the premisses of the forward approach. Moving backwards through the code these will be transformed according to standard weakest precondition rules. If the transformed proof obligations do not hold in the program's initial state, a security violation is flagged.

Both rules apply a standard technique, *sp* or *wp*, to reason over the line of code and additionally add conditions required for security. These conditions must hold in the state before

$$wpif(x := e, Q) \mathrel{\widehat{=}} \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge (x \in \mathcal{C} \Rightarrow secure\_update(x, e)) \\ \wedge\ wp(x, \Gamma_x := e, \Gamma_E(e), Q)$$

where $secure\_update(x, e)$ is defined as in Figure 2.

Fig. 3: Backwards assignment rule (based on [5]).

the assignment is executed. To encode the rules in Dafny, which already performs *wp* reasoning, we simply add the required conditions as an assertion before each assignment. To do this, we (i) introduce an *auxillary variable*, i.e., one that does not affect the program behaviour, representing $\Gamma(x)$ for each variable x, and (ii) calculate $\Gamma_E(e)$ as the highest security level of any value held by a variable v that appears free in e. It is possible that the value of $\Gamma(v)$ cannot be deduced, e.g., when v is uninitialised or an input, in which case Dafny will allow any value. In such cases, we restrict the value to $\mathcal{L}(v)$ which is the highest value which it would hold if the program is secure up to the point where the assignment is executed. Hence, $\Gamma_E(e) \mathrel{\widehat{=}} \bigsqcup_{v \in vars(e)} (\Gamma(v) \sqcap \mathcal{L}(v))$ where $vars(e)$ returns the free variables in e.

Finally, for the *secure_update* predicate we need to be able to evaluate $\mathcal{L}(y)$ for a controlled variable y with the control variable x replaced by e. This can be done using Dafny's notation for optional parameters to functions. For example, the function L_x above could be rewritten as follows

```
function L_x(a:int := z):L
   reads this
{if a == 0 then B else D}
```

where a is an optional integer parameter which, when not explicit in the function call, defaults to z.

Assume we have a class in which x, y and z of type int have been declared along with their associated auxiliary variables Gamma_x, Gamma_y and Gamma_z of type L. Assume also that there are functions L_x, L_y and L_z defined in the class to return the value-dependent security classification of variables x, y and z, respectively. Given that L_x is defined as above, we would require the following assertion to hold before an assignment z := x + y occurring in the method representing the program.

```
assert leq(join(meet(Gamma_x, L_x()),
                meet(Gamma_y, L_y())), L_z()) &&
       leq(meet(Gamma_x, L_x()), L_x(x + y));
```

The first conjunct of the assertion calculates $\Gamma_E(x+y)$ and ensures that it is not greater than $\mathcal{L}(z)$ (ensuring no leak of information through z), and the second calculates $\Gamma_E(x)$, i.e., the security level of the data held by x, and ensures that it is not greater than $\mathcal{L}(x)[z \backslash x+y]$ (ensuring no leak of information through x). This assertion can be derived directly from the syntax of the assignment and hence can be readily automated.

We also require the assignment to include the simultaneous update of Γ_z as below.[1]

```
z, Gamma_z := x+y, join(meet(Gamma_x, L_x()),
                        meet(Gamma_y, L_y()));
```

[1]Note that calls to functions from code are allowed since Dafny 4.0.

This rewriting of the assignment z := x + y can again be readily automated.

B. Local variables and inputs and outputs

Handling assignments to local variables and inputs and outputs requires modification to the above encoding. Local variables do not need a security classification; they are not accessible outside the program and hence may hold data at any security level. However, since they may be assigned to other program variables we must track the level of information they hold with a Γ variable. Hence, we add an auxiliary variable and update the assignment statement as in the previous section, but do not require any assertions to be checked. (Note that local variables cannot be in $\mathcal{C}$.) Furthermore, in the absence of explicit initialisation we assume that a local variable will be initialised to either a default or arbitrary value. Hence, when we declare the local variable we assume it holds no classified information and set its Γ variable to the lowest security level as follows (where D is the lowest security level as in the example lattice defined previously).

```
var y: int;
var Gamma_y := D;
```

Input variables have a fixed value throughout a program. Hence, we restrict the security level of the value they hold in the precondition (requires clause) of the program. To do so, for each input variable a, we introduce an auxiliary input variable Gamma_a. Output variables are like local variables except they become accessible outside the program when it terminates. Hence, we restrict the security level they hold in the program's postcondition (ensures clause). Again this requires the addition of an auxiliary variable Gamma_x for any output x. For example, the signature of the Mult method of Figure 1a would be (automatically) transformed to

```
method Mult(a:int, Gamma_a:L, b:int, Gamma_b:L)
        returns (x:int, Gamma_x:L)
```

C. Arrays

Following [22], we require that the security level of an array a is at least as high as that of (i) any element in a (if an attacker can access the array, they can access any element in the array) and (ii) any index i used in an assignment a[i] := v (if an attacker can subsequently read an element at index j, they can determine whether (or not) i is equal to j). That is, for the assignment a[i] := x we require the assertion

```
assert leq(join(meet(Gamma_x, L_x()),meet(Gamma_i, L_i())),
           L_a())
```

and for Gamma_a to be updated to

```
join(join(meet(Gamma_x, L_x()), meet(Gamma_i, L_i())),
     meet(Gamma_a, L_a()))
```

Note that the security level of a may be higher than that of both x and i and hence is updated to be the join of these three security levels.

In some programs, arrays may need to simultaneously hold information at different security levels. Hence in addition to

the above, we treat each array element as a separate variable with its own security classification. To do this in Dafny, for an array a we add an array Gamma_ai of the same length as a, and a function L_ai taking a single parameter corresponding to the position in the array as follows.

```
function L_ai(i:nat):L
   requires i < a.Length
   reads this
...
```

For assignment a[i] := x, we check leq(meet(Gamma_x,L_x()), L_ai[i]) and update Gamma_ai[i] to meet(Gamma_x, L_x()).

When reading an element a[j], we calculate its security level as for a variable, i.e., it will be meet(Gamma_ai[j],L_ai(j)), except when an element of the array has been updated using an index i of a higher security level. In that case, we must assign the security level of i to a[j] as the value of a[j] can be used to deduce information about i (case (ii) above).

Hence, we add a variable minL_ai:L which is initialised to the lowest security level of the security lattice and updated to join (meet(Gamma_i,L_i()),minL_ai) at each assignment a[i] := x. When a[j] is read, its security level is calculated as the maximum of that of the value it currently holds and minL_ai, i.e., join (meet(Gamma_ai[j],L_ai(j)),minL_ai).

Finally, particular array values may be used as control variables. In this case, we cannot use the optional parameter approach in Section III-A (as an expression a[i], for some i, cannot be used in the parameter list since the length of a is not defined there). Instead, we require a second function which takes a parameter corresponding to the value being assigned to a[i]. For example, if the security classification of a variable x depends on a[i], for some i, we could declare the following function, to be used for the *secure_update* condition when we are assigning a value v of type T to a[i].

```
function L_xv(v:T):L
  ...
```

D. Branching statements

Information can also be leaked by branches in the program (belonging to if statements and loops). This occurs when classified information is used within the branch guard and differences in timing of the branches allows an attacker to deduce which branch was taken and hence deduce the classified information [23], [24]. Therefore, the rules for if statements and loops in Murray et al. [4] and Winter et al. [5] require that $\Gamma_E(b)$, where b is the guard, is not higher than the security level that a potential attacker can observe. For multi-level lattices, this level needs to be the bottom of the lattice to ensure no attacker, regardless of the level they can observe, can deduce classified information. Rules, based on those in Winter et al. [5], are shown in Figure 4.

Again, standard reasoning, already supported by Dafny, is augmented with the additional condition for security. Hence, the rules can be captured by an assertion before the guard of the if statement or loop. For example, given D is the lowest security level then before a statement if x < y ... we would have

$$
\begin{aligned}
&\mathit{wpif}(\textsf{if } b\ \{c_1\}\ \textsf{else}\ \{c_2\}, Q) \mathrel{\widehat{=}} \\
&\qquad \Gamma_E(b) \sqsubseteq \bot \wedge \\
&\qquad (b \Rightarrow \mathit{wpif}(c_1, Q)) \wedge (\neg\, b \Rightarrow \mathit{wpif}(c_2, Q))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{wpif}(\textsf{while } b\ \{c\}, Q) \mathrel{\widehat{=}} \\
&\qquad \mathit{Inv} \wedge (\forall\, \sigma \cdot \Gamma_E(b) \sqsubseteq \bot) \wedge \\
&\qquad \forall\, \sigma \cdot (\mathit{Inv} \wedge b \Rightarrow \mathit{wpif}(c, \mathit{Inv})) \wedge (\mathit{Inv} \wedge \neg\, b \Rightarrow Q)
\end{aligned}
$$

where $\bot$ is the lowest level of the security lattice, *Inv* is a user-provided loop invariant, and σ denotes an instance of the program state.

Fig. 4: Conditional and loop rules (based on [5]).

```
assert leq(join(meet(Gamma_x, L_x()), meet(Gamma_y, L_y())), D);
```

Again the inclusion of this assertion is readily automated.

E. Declassification

In practice, *declassification*, i.e., controlled release of classified information, is required in many programs. For example, a password checker revealing that a guessed password is wrong releases some information about the password. Hence, information flow analyses need to support a notion of declassification [25].

Smith [26] proposes a declassification approach which allows a programmer to state precisely *what* may be declassified and *where* in the program the declassification may occur. The approach uses a predicate, defined by the programmer, at the point of declassification to relate the expression being declassified to values in the program's initial state. Only when the predicate is true is the declassification allowed. Such a predicate can be encoded as a two-state predicate in Dafny.

When the user provides the predicate *true*, the approach allows unconditional declassification of information at a point in the program similar to approaches such as that of Mantel and Sands [27]. When they choose any other predicate, the approach captures not only the point where declassification occurs, but what information is allowed to be declassified. Similar to approaches such as those of Sabelfeld and Myers [28] and Askarov and Sabelfeld [29], this enables the detection of programmer errors which could lead to information leaks.

For example, a program which updates a classified integer variable x to a new value v may be allowed to release the change in the value of the x, but not its new or old value. This can be encoded as follows.

```
twostate predicate P(y:int, v:int) {y == old(x) – v};

method M(v:int)
   modifies this
{
   ...
   assert P(x – v, v);
   diff := x – v;
   x := v;
}
```

Note that if a programmer mistakenly included x := 0; in the elided part of the method, the assertion would fail (since old(x)

is not necessarily 0), and hence the leak of x's new value through diff would be detected. Although this simple example does not allow any change to x before the declassification point, the predicate P can allow a range of values for its input if a change to x is required (see [26] for examples).

F. Information flow in other program verifiers

While we can expect assertions and functions to be supported by program verifiers other than Dafny, our approach above relies on notation that is more Dafny-specific. It includes predicates, simultaneous assignments, optional parameters, classes and two-state predicates. It is trivial, however, to get by without each of these: predicates can be replaced by Boolean-valued functions, and the other notation can be captured using additional local variables and parameters. For example, a two-state predicate can be captured using a predicate (or Boolean-valued function) with parameters corresponding to the values of the variables in the old state. These parameters would be instantiated with local variables that have been set to the state variables at the point in the program corresponding to the desired old state.

Only lemmas cannot be represented readily in other tools that do not already support a similar construct. These are useful in our approach to detect mistakes made by the user when defining the security lattice. When using other tools these simple proofs could be done using an external theorem prover, or alternatively using the Dafny verifier.

IV. Rely/guarantee reasoning

Rely/guarantee reasoning [7], [8] is a general method for thread-local analysis of concurrent programs and has been used in a number of information flow analysis approaches [30], [3], [4], [9], [5]. Each thread i is given a *rely* condition, $\mathcal{R}_i$, which models the behaviour of the other threads in the program. When reasoning about i we rely on the fact that all interference, occurring between i's program steps, conforms to $\mathcal{R}_i$. For this to be sound, each thread also has a guarantee condition, $\mathcal{G}_i$, which each of its program steps conform to, and which implies the rely conditions of all other threads, i.e., $\forall\, i \cdot G_i \Rightarrow \bigwedge_{j \neq i} R_j$. We refer to this condition as *compatibility*.

The rely and guarantee conditions are relations over pre- and post-states and must be reflexive (corresponding to the possibility that no steps are taken or only steps that do not change the shared state). Additionally, rely conditions must be transitive (and therefore independent of the number of steps taken).

Although Dafny does not support concurrency, we can use rely/guarantee reasoning to reason about a sequential program as if it were a thread in a concurrent program. Each such thread can be modelled by a separate method in a class representing the entire program. The associated rely and guarantee predicates can be modelled by two-state predicates provided by the user. Reflexivity and transitivity of these can be checked using auxiliary methods. Consider the following in which R_1 and G_1 are the rely and guarantee condition of a particular thread, respectively.

```
twostate predicate R_1()
  reads this
...

twostate predicate G_1()
  reads this
...

method ReflexiveR_1()
  ensures R_1()
{}

method ReflexiveG_1()
  ensures G_1()
{}

method Rely_1()
  modifies this
  ensures R_1()

method TransitiveR_1()
  modifies this
  ensures R_1()
{
  Rely_1();
  Rely_1();
}
```

The methods ReflexiveR_1 and ReflexiveG_1 will only verify in Dafny when the two-state predicates R_1 and G_1, respectively, hold for an empty method body in which there is no change to the program state. Hence, they check that the supplied conditions are reflexive. The method Rely_1 has a specification, but no body. Since method calls are reasoned about in Dafny using the method's specification only, a call to Rely_1 models any update corresponding to R_1. It can be used in the final method TransitiveR_1 which will only verify in Dafny when two successive calls to Rely_1() satisfy R_1. This is sufficient to prove that R_1 is transitive. As with the lemmas for checking the security lattice properties, these auxiliary methods will detect mistakes made by the user when defining the rely and guarantee conditions.

With these definitions in place, we can automatically update the method corresponding to the thread to (i) include a call to Rely_1 at the start of the method and after each instruction (modeling the threads in the environment taking zero or more steps) and (ii) check that each assignment step of the program satisfies G_1. (Other steps, such as local variable declarations and checking of conditional and loop guards, do not change the program variables and hence trivially satisfy any reflexive guarantee. Method calls are discussed in Section VI.) As is usual for rely/guarantee reasoning, update (i) requires that the instructions of the thread are atomic. When there are non-atomic instructions, e.g., x := x+1, these instructions need to be decomposed into atomic steps, e.g., x := x+1 becomes var y := x; x := y+1. Update (ii) requires that a label is added to each assignment of the method. For example, consider a thread with the rely and guarantee conditions above which updates an integer variable x and then an integer variable y neither of which are control variables. The required Dafny method corresponding to this thread is shown below (the assertions correspond to the associated assignment proof obligations).

```
method T_1()
  modifies this
{
  Rely_1();
  assert leq(meet(Gamma_a, L_a()), L_x());
  label 1: x, Gamma_x := a, meet(Gamma_a, L_a());
  assert G_1@1();
  Rely_1();
  assert leq(meet(Gamma_b, L_b()), L_y());
  label 2: y,Gamma_y := b, meet(Gamma_b, L_b());
  assert G_1@2();
  Rely_1();
}
```

We also need to check compatibility. If this thread occurs in a program with two other threads whose rely conditions are captured by two-state predicates R_2 and R_3, then we also include a Dafny two-state lemma checking that G_1 is strong enough to satisfy these conditions. This lemma is independent of the threads' code and can be automatically generated.

```
twostate lemma CompatibleG_1()
  ensures G_1() ==> R_2() && R_3()
{}
```

In practice, the guarantee of a thread is often chosen to be the conjunction of the rely conditions of the other threads in the program. In such cases, this lemma can be readily verified without a user-provided proof.

A. Rely/guarantee reasoning in other program verifiers

Dafny-specific notation introduced for rely/guarantee reasoning includes the use of labels with two-state predicates and methods without a body (used for method Rely_1 in the example). The former can be handled as described in Section III-F. While bodiless methods may not be supported in other verifiers, such verifiers will adopt a compositional approach to reasoning in which method calls are verified in terms of the called method's specification, not its code. Hence, an arbitrary body that satisfies the specification can be provided. For a reflexive method such as Rely_1, an empty body can be used.

The sharing of variables between methods in a class, while convenient in our approach, is not strictly required as each thread is verified in terms of the rely conditions of the other threads, not the threads themselves. Hence, tools without support for classes can simply introduce the shared state as local variables in each thread.

V. Dynamic thread creation and termination

Rely/guarantee reasoning assumes all threads are running when the program starts. In reality, threads are created and terminated dynamically as a program executes. Feng and Shao [31] generalise rely/guarantee reasoning to allow dynamic thread creation and termination facilitating reasoning about realistic concurrent programs. When a thread T_1 forks a new thread T_2, to maintain compatibility that new thread must not require more from T_1's environment than T_1 required, i.e., $R_1 \Rightarrow R_2$. Also, T_1's guarantee must ensure R_2 and hence is updated to be $G_1 \wedge R_2$ while T_2 is potentially still running, i.e., until T_2 joins. In turn, T_2 must guarantee at least what T_1 was guaranteeing to its environment, i.e., $G_2 \Rightarrow G_1$. Also, T_1 must include T_2 as part of its environment while T_2 is potentially still running and hence its rely condition is updated to $R_1 \vee G_2$. These conditions are summarised in Figure 5.

When thread $T1$ forks thread $T2$ we require

- $R_1 \Rightarrow R_2$
- $G_2 \Rightarrow G_1$
- $T1$'s rely condition is updated to $R_1 \vee G_2$
- $T1$'s guarantee condition is updated to $G_1 \wedge R_2$

Fig. 5: Thread creation rules (based on [31]).

Although not discussed in [31], it is necessary that the updated guarantee is reflexive, and the updated rely condition is both reflexive and transitive. It is easy to see that the conjunction of two reflexive predicates is also reflexive, i.e., if the state transition (s, s) is allowed under G_1 and also under R_2 then it is allowed under $G_1 \wedge R_2$. Similarly, it is easy to see that the disjunction of two reflexive predicates is reflexive, i.e., if (s, s) is allowed under R_1 and also under G_2 then it will be allowed under $R_1 \vee G_2$. The disjunction of R_1 and G_2, however, will not always be transitive. For example, if R_1 allows the single transition (s_1, s_2) it will be trivially transitive (as at most one step can be taken). If G_2 allows the transition (s_2, s_3) but not (s_1, s_3) then $R_1 \vee G_2$ will fail to be transitive, allowing (s_1, s_2) and (s_2, s_3) but not (s_1, s_3).

Hence, it is necessary to check that $R_1 \vee G_2$ is transitive each time we fork a thread. In the case that it is not transitive, the rely will need to be weakened by the user based on their understanding of the program. Weakening (rather than strengthening) the rely, will ensure other properties, such as compatibility of the forking thread with its environment, are not violated.

To model forking and joining of threads in Dafny, we introduce a pair of methods, $Fork_i$ and $Join_i$, for each thread i. These methods have no specification or body and just act as a marker for where the rely and guarantee conditions of the calling thread need to be updated. The fork method additionally introduces a proof obligation that the new thread's rely and guarantee conditions meet the requirements above. For example, consider the example thread T_1 of Section IV where additionally a second thread T_2 with rely and guarantee conditions R_2 and G_2 respectively, is forked and joined. We introduce a new method corresponding to the updated rely condition for T_1

```
method Rely_12()
  modifies this
  ensures R_1() || G_2()
```

and the following lemma and method to verify the required conditions.

```
lemma ForkConditions_12()
  ensures R_1() ==> R_2()
  ensures G_2() ==> G_1()
{}
```

```
method TransitiveR_12()
   modifies this
   ensures R_1() || G_2()
{
   Rely_12();
   Rely_12();
}
```

The required Dafny method corresponding to the thread is then as shown below where the method calls Fork_2() and Join_2() mark where the thread T_2 forks and joins, respectively.

```
method T_1()
   modifies this
{
   Rely_1();
   assert leq(meet(Gamma_a, L_a()), L_x());
   label 1: x, Gamma_x := a, meet(Gamma_a, L_a());
   assert G_1@1();
   Rely_1();
   Fork_2();
   Rely_12();    // updated rely condition
   assert leq(meet(Gamma_b, L_b()), L_y());
   label 2: y, Gamma_y := b, meet(Gamma_b, L_b());
   assert G_1@2() && R_2@2();    // updated guarantee
   Rely_12();
   Join_2();
   Rely_1();   // return to original rely (and guarantee) condition
}
```

The methods Fork_2 and Join_2 are defined without a body and modifies clause and hence do not change the global state.

```
method Fork_2()

method Join_2()
```

A. Thread creation and termination in other program verifiers

In the unlikely case, that another program verifier does not have something equivalent to a modifies clause, the postcondition of the fork and join methods would need to state that every shared variable is unchanged. No additional Dafny-specific notation is otherwise introduced in this section.

VI. Method calls

Program verifiers generally reason about method calls using the specification of the method rather than (inlining) its code. This allows the called method's code to be reasoned about only once (to see that it meets its specification), and recursive and non-recursive method calls to be treated uniformly.

For information flow analysis, we require that the specification (pre- and postcondition) of a method called from a thread captures any changes to the security values held by shared variables, and the values of the variables themselves when they are control variables (as such changes may affect the security classifications of other variables in the program).

Note that assignments of return values will be to output variables of the method, followed by an assignment to shared or local program variables in the calling code when required. This allows the calling code's guarantee to be checked on the latter (and also captures the fact that the call and subsequent assignment is not atomic). To ensure the called method M also satisfies the calling code's guarantee, we treat it like a new thread with its own rely and guarantee conditions, R_M and G_M, respectively. For each call from a thread T_i, we add a lemma (like that for forking a thread) to check that $R_i \Rightarrow R_M$ and that $G_M \Rightarrow G_i$. While R_M and G_M must be provided by the programmer, the lemma can be generated automatically.

VII. Case study: producer-consumer

This section presents a case study of a producer-consumer program (designed to illustrate the main concepts in this paper). For simplicity of presentation, we assume all assignments and guards are atomic. The program comprises a Main thread which forks a Producer and a Consumer thread.

```
class ProducerConsumer {
   method Fork_Producer()
   method Fork_Consumer()

   method Main() {
      Fork_Producer();
      Fork_Consumer();
   }
   ...
}
```

The producer repeatedly gets a new integer value and a boolean, indicating whether that value is classified, by calling a method GetValue. The new value is added to the tail position of a (sufficiently large) finite buffer, values. The tail position is then incremented indicating to the consumer that there is a new element.

The consumer may only consume unclassified values. Hence, when a classified value is received by the producer, it does not increment tail but forks a HighConsumer thread to consume the value and waits for this thread to join.

```
const values:array<int>
var head:nat
var tail:nat

method GetValue() returns (v:int, c:bool)

method Fork_HighConsumer()
method Join_HighConsumer()

method Producer()
   modifies this, values
   decreases *
{
   var v:int;
   var c:bool;
   while tail < values.Length
      decreases *
   {
      v,c := GetValue();
      values[tail] := v;
      if c == true {
         Fork_HighConsumer();
         Join_HighConsumer();
      } else {
         tail := tail + 1;
      }
   }
}
```

Note that the decreases clause (used for the loop's variant in Dafny) is * to indicate that the loop does not necessarily terminate.

The thread forked by the producer thread processes the value at the buffer's tail by calling the method HighProcess. It also stores the value in a variable classified and declassifies the difference between the current value and that previously stored.

```
method HighProcess(v:int)

var classified:int
var diff:int

method HighConsumer()
   requires tail < values.Length
   modifies this
{
   var temp := values[tail];
   HighProcess(temp);
   diff := temp – classified;  // declassification
   classified := temp;
}
```

The consumer thread repeatedly checks if head is less than tail , i.e., if there is an element in the buffer, and if so calls the method Process to process the integer at the buffer's head. It then increments head to remove the value from the buffer.

```
method Process(v:int)

method Consumer()
   requires tail <= values.Length
   modifies this
   decreases *
{
   while true
      invariant tail <= values.Length
      decreases *
   {
      if head < tail {
         Process(values[head]);
         head := head + 1;
      }
   }
}
```

A. User-defined security policy

For this program to be secure, we require that all values that can be read by the consumer thread, i.e., all elements of values between positions head and tail are not classified.

To check this using the approach from this paper we define a boolean security lattice (since values are either classified or not).

```
datatype L = Low | High

predicate leq(l1:L, l2:L) { l1 == Low || l2 == High }

function join(l1:L, l2:L):L { if leq(l1, l2) then l2 else l1 }

function meet(l1:L, l2:L):L { if leq(l1, l2) then l1 else l2 }
```

This lattice satisfies the required lemmas from Section III.

We can then specify the security classification for values and elements of values as follows.

```
function L_values():L { High }

function L_valuesi(i:nat, h:int := head, t:int := tail):L
   requires i < values.Length
   reads this
{
   if h <= i < t then Low else High
}
```

Note that this makes head and tail control variables.

The security classifications of head, tail and diff are Low in any state, and that of classified is High in any state. Hence, we have

```
function L_head():L {Low}
function L_tail():L {Low}
function L_diff():L {Low}
function L_classified():L {High}
```

We also require a declassification predicate for the information released through diff . This predicate is

```
twostate predicate P(y:int)
   requires old(tail) < values.Length
   reads this
{ y == old(values[tail]) – old(classified) }
```

B. Used-defined rely/guarantee conditions

The programmer also needs to provide the rely and guarantee conditions for each thread and each method called from a thread. For the Main thread we simply let the rely condition be the conjunction of those for the threads it forks and the guarantee be true (satisfying the conditions of Section V).

The producer thread can place classified values in the buffer. To ensure the security policy, it relies on no other thread changing tail as this may result in the classified value being in the range for which L_valuesi only allows Low values. It must also rely on other threads not updating diff or classified while the high consumer thread it forks is running.

```
twostate predicate R_Producer()
   reads this
{ tail == old(tail) && diff == old(diff) && classified == old(classified) }
```

Its guarantee is equal to the consumer thread's rely condition defined below.

```
twostate predicate G_Producer()
   reads this
{ R_Consumer() }
```

The method GetValue called by the producer thread does not rely on anything from the other threads (we assume it operates independently of the program variables).

```
twostate predicate R_GetValue() {true}
```

Its guarantee is equal to that of the producer (trivially satisfying the condition that it implies the latter).

The rely and guarantee conditions for the HighConsumer thread are equal to those of the producer (trivially satisfying the required conditions).

Like GetValue, we assume the HighProcess and Process methods do not rely on the program variables and hence have a true rely condition, and that their guarantees are equal to those of their calling threads.

```
twostate predicate R_HighProcess() {true}

twostate predicate R_Process() {true}
```

Since the Consumer thread passes the buffer value at head to the method Process (which requires a Low value), it requires that its environment does not change head, and only increases tail while maintaining the conditions tail <= values.Length (required for its loop invariant) and minL_valuesi == Low.

```
twostate predicate R_Consumer()
  reads this
{
  head == old(head) && old(tail) <= tail &&
  (old(tail) <= values.Length ==> tail <= values.Length) &&
  (old(minL_valuesi) == Low ==> minL_valuesi == Low)
}
```

Its guarantee is equal to the producer's rely condition defined above.

All of the above rely and guarantee conditions are reflexive and all of the rely conditions are transitive as required.

C. Automatically generated annotations

Once the programmer has defined the security policy and rely and guarantee conditions, the auxiliary variables and methods, assertions and lemmas required to verify security can be automatically added to the program. For each called method, auxiliary variables for inputs and outputs are added. For example, Process is updated as follows.

```
method Process(v:int, Gamma_v:L)
```

For each thread, required assertions, initialisation of local variables, calls to the associated rely method and checking of the associated guarantee are added. For example, the assignment head := head + 1; of the consumer thread is updated as follows.

```
assert leq(meet(Gamma_head, L_head()), L_head()) &&
       forall j:: 0 <= j < values.Length ==>
           leq(meet(Gamma_valuesi[j],L_valuesi(j)),
               L_valuesi(j,head+1));
label Consumer1: head, Gamma_head :=
                 head + 1, meet(Gamma_head,L_head());
assert G_Consumer@Consumer1();
Rely_Consumer();
```

D. User-defined conditions

Next, the user needs to provide pre and postconditions for methods that are called. The GetValue method must return a consistent pair of outputs, i.e., GetValue() is updated to

```
method GetValue() returns (v:int, Gamma_v:L, c:bool, Gamma_c:L)
  ensures (Gamma_v == High <==> c == true) &&
          Gamma_c == Low
```

The HighProcess method does not constrain its input and hence does not need to be updated. The Process method, on the other hand, requires its input to be Low and is updated accordingly.

```
method Process(v:int, Gamma_v:L)
  requires Gamma_v == Low
```

The user also needs to weaken any modified rely conditions of forking threads that are not transitive. For example, the rely generated after the Consumer thread is forked, (R_Main || G_Producer) || G_Consumer, needs to be weakened by removing the conjunct head == old(head) from G_Producer.

Finally, program invariants are added as preconditions to each thread and loop invariants are updated as required. For the case study, the precondition tail <= values.Length && minL_valuesi == Low && Gamma_valuesi.Length == values.Length is added to each thread, and the invariant tail <= values.Length && minL_valuesi == Low to each loop. The need for these additional constraints is determined as they would be in a standard Dafny development. With these changes in place, the full program can be proven secure using the Dafny verifier.

VIII. Conclusion

This paper has shown that program verifiers like Dafny, which were built for verifying functional correctness of sequential programs, can also be used for verifying information flow security for concurrent programs. Our approach captures the information flow rules of the state-or the-art approaches by Murray et al. [3], [4] and Winter et al. [5] which utilise compositional rely/guarantee reasoning for thread-local analysis. It also extends these approaches to handle a number of constructs that commonly arise in real software: arrays, information declassification, dynamic thread creation and termination, and method calls.

The approach transforms a Dafny program by adding (i) a security policy, comprising a security lattice, security classifications and declassification predicates, (ii) rely and guarantee conditions for each thread, and (iii) assertions, auxiliary variables and methods, and lemmas needed to verify security. The latter, based on the program syntax, can be automatically generated by a front-end transpiler.

For the approach to be practical for larger programs, tool support is essential. A prototype tool covering basic information flow (not arrays nor declassification) and basic rely/guarantee reasoning (not dynamic thread creation) has been developed as an extension to Dafny's source code (https://github.com/dafny-lang/dafny). The tool runs in two phases. The first phase generates skeleton code for the security policy and rely and guarantee conditions which the user fills in before running the second phase. The second phase generates the assertions and other annotations required to verify security based on the program code. The output of the second phase can be modified further, e.g., to add method pre and postconditions, as required.

The prototype tool also includes a way of modelling an atomic compare-and-swap (CAS) operator in Dafny. This allows threads to use locks, a common synchronization mechanism in concurrent programs. The information flow checks required when using a CAS operator can be found in [5].

Future work will consider using Boogie directly to reason about unstructured assembly code, following the ideas in [32]. This will allow the analysis to account for any optimisations to the code introduced during compilation (a well known source of security leaks) [33]. Also, approaches to automatically generating security policies for value-dependent information flow analysis [34] and rely/guarantee conditions for general concurrent programs [35] will be investigated to further lessen the burden on the programmer when using our approach.

Ackowledgements Thanks to Daniel Parkinson and Sanni Bosamia for their contributions to the ideas in this paper.

REFERENCES

[1] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, no. 2/3, pp. 167–188, 1996.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003.

[3] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016*. IEEE Computer Society, 2016, pp. 417–431.

[4] T. C. Murray, R. Sison, and K. Engelhardt, "COVERN: A logic for compositional verification of information flow control," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. IEEE, 2018, pp. 16–30.

[5] K. Winter, N. Coughlin, and G. Smith, "Backwards-directed information flow analysis for concurrent programs," in *34th IEEE Computer Security Foundations Symposium, CSF 2021*. IEEE, 2021, pp. 1–16.

[6] G. Ernst and T. Murray, "SecCSL: Security Concurrent Separation Logic," in *Computer Aided Verification - 31st International Conference, CAV 2019*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11562. Springer, 2019, pp. 208–230.

[7] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983, pp. 321–332.

[8] Q. Xu, W. P. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *Formal Aspects of Computing*, vol. 9, no. 2, pp. 149–174, 1997.

[9] N. Coughlin and G. Smith, "Rely/guarantee reasoning for noninterference in non-blocking algorithms," in *33rd IEEE Computer Security Foundations Symposium, CSF 2020*. IEEE, 2020, pp. 380–394.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[11] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[12] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 364–387.

[13] J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds., vol. 7792. Springer, 2013, pp. 125–128.

[14] G. Parthasarathy, P. Müller, and A. J. Summers, "Formally validating a practical verification condition generator," in *Computer Aided Verification - 33rd International Conference, CAV 2021*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 704–727.

[15] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370.

[16] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[17] S. Blom and M. Huisman, "The VerCors tool for verification of concurrent programs," in *FM 2014: Formal Methods - 19th International Symposium*, ser. Lecture Notes in Computer Science, C. B. Jones, P. Pihlajasaari, and J. Sun, Eds., vol. 8442. Springer, 2014, pp. 127–131.

[18] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The Prusti project: Formal verification for Rust," in *NASA Formal Methods - 14th International Symposium, NFM 2022*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 88–108.

[19] X. Denis, J. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of Rust programs," in *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022*, ser. Lecture Notes in Computer Science, A. Riesco and M. Zhang, Eds., vol. 13478. Springer, 2022, pp. 90–105.

[20] K. R. M. Leino and N. Polikarpova, "Verified calculations," in *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Springer, 2013, pp. 170–190.

[21] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, 1982*. IEEE Computer Society, 1982, pp. 11–20.

[22] G. Barany, "Hybrid information flow analysis for programs with arrays," in *Proceedings of the Fourth International Workshop on Verification and Program Transformation, VPT@ETAPS 2016*, ser. EPTCS, G. W. Hamilton, A. Lisitsa, and A. P. Nemytykh, Eds., vol. 216, 2016, pp. 5–23.

[23] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium, USENIX Security 16*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 53–70.

[24] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology - ICISC 2005, 8th International Conference*, ser. Lecture Notes in Computer Science, D. Won and S. Kim, Eds., vol. 3935. Springer, 2005, pp. 156–168.

[25] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, 2009.

[26] G. Smith, "Declassification predicates for controlled information release," in *23rd International Conference on Formal Engineering Methods (ICFEM 2022)*, ser. Lecture Notes in Computer Science, A. Riesco and M. Zhang, Eds. Springer, 2022.

[27] H. Mantel and D. Sands, "Controlled declassification based on intransitive noninterference," in *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, ser. Lecture Notes in Computer Science, W. Chin, Ed., vol. 3302. Springer, 2004, pp. 129–145.

[28] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, ser. Lecture Notes in Computer Science, K. Futatsugi, F. Mizoguchi, and N. Yonezaki, Eds., vol. 3233. Springer, 2003, pp. 174–191.

[29] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007*, M. W. Hicks, Ed. ACM, 2007, pp. 53–60.

[30] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*. IEEE Computer Society, 2011, pp. 218–232.

[31] X. Feng and Z. Shao, "Modular verification of concurrent assembly code with dynamic thread creation and termination," in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, O. Danvy and B. C. Pierce, Eds. ACM, 2005, pp. 254–267.

[32] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05*, M. D. Ernst and T. P. Jensen, Eds. ACM, 2005, pp. 82–87.

[33] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015*. IEEE Computer Society, 2015, pp. 73–87.

[34] P. Li and D. Zhang, "A derivation framework for dependent security label inference," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 115:1–115:26, 2018.

[35] X. Le, D. Sanán, J. Sun, and S. Lin, "Automatic verification of multithreaded programs by inference of rely-guarantee specifications," in *25th International Conference on Engineering of Complex Computer Systems, ICECCS 2020*, Y. Li and A. W. Liew, Eds. IEEE, 2020, pp. 43–52.