Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security

Marco Eilers, Severin Meier, and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
marco.eilers@inf.ethz.ch, sev.meier@gmail.com,
peter.mueller@inf.ethz.ch

Abstract. Most existing program verifiers check trace properties such as functional correctness, but do not support the verification of hyperproperties, in particular, information flow security. In principle, product programs allow one to reduce the verification of hyperproperties to trace properties and, thus, apply standard verifiers to check them; in practice, product constructions are usually defined only for simple programming languages without features like dynamic method binding or concurrency and, consequently, cannot be directly applied to verify information flow security in a full-fledged language. However, many existing verifiers encode programs from source languages into simple intermediate verification languages, which opens up the possibility of constructing a product program on the intermediate language level, reusing the existing encoding and drastically reducing the effort required to develop new verification tools for information flow security.
In this paper, we explore the potential of this approach along three dimensions: (1) Soundness: We show that the combination of an encoding and a product construction that are individually sound can still be unsound, and identify a novel condition on the encoding that ensures overall soundness. (2) Concurrency: We show how sequential product programs on the intermediate language level can be used to verify information flow security of concurrent source programs. (3) Performance: We implement a product construction in Nagini, a Python verifier built upon the Viper intermediate language, and evaluate it on a number of challenging examples. We show that the resulting tool offers acceptable performance, while matching or surpassing existing tools in its combination of language feature support and expressiveness.

1 Introduction

Since computer programs increasingly handle sensitive user data and communicate using encryption, it is vital that programs do not leak secret data such as private keys to attackers, that is, that they are *information flow secure*. One way of formalizing information flow security is *noninterference*, a so-called *2-hyperproperty*, i.e., a property of pairs of executions of the program.

Noninterference can be checked by type systems [45] and static analyses [23]. However, complex language features (such as concurrency) and noninterference properties (such as termination sensitivity) generally require the expressiveness of deductive verification. In recent years, many automated and expressive verification tools have been developed for a wide range of programming languages, but most of these tools are limited to trace properties (properties of single program traces) and cannot prove hyperproperties such as noninterference.

The problem we address in this paper is how to retrofit existing program verifiers to check noninterference. Compared to building noninterference verifiers from scratch, which can take years when targeting substantial subsets of real-world programming languages, this approach would allow us to reuse most aspects of existing verifiers, such as the semantic representation of language features and proof search algorithms. Moreover, it naturally allows one to verify combinations of correctness and noninterference properties.

In principle, existing program verifiers can be used to verify hyperproperties by reducing them to trace properties via self-composition [6] or product programs [4, 5]. However, self-composition does not allow modular verification [48], and product programs have generally been defined only for simple languages without features like dynamic method binding or concurrency [4, 19]. Applying product constructions to programs written in complex languages would therefore require defining and implementing new and complex product constructions for every new verifier.

We explore a more efficient approach here: We leverage the fact that most automatic deductive verifiers are organized into a custom frontend, which encodes a source program into an intermediate verification language (IVL), and a reusable backend, which verifies the IVL program using generic proof search engines. Boogie [3], Viper [35], and Why3 [22] are examples of such IVLs, which power a large number of program verifiers; for instance Boogie is used by Dafny [30], VCC [13], Spec# [31], and GPUVerify [8], Why3 [22] by Frama-C [14] and Krakatoa [21], and Viper [35] by Vercors [10], Prusti [2], and Nagini [18]. The ubiquitiy of this architecture offers a chance to retrofit existing verifiers to check noninterference by performing the product construction on the level of the IVL (an approach that is already used by SymDiff [28] for the related problem of program equivalence). The resulting architecture, which allows one to reuse both the frontend and the backend of the existing verifier, is shown in Fig. 1.

Performing the product construction on the IVL-level has three major advantages over a product construction on the source program: (1) It cleanly separates the encoding of the source language (which tends to be complex for full-fledged languages) from the product construction. (2) The product construction is much simpler since IVLs are small, sequential languages. (3) The product construction can be reused across all verifiers built on the same IVL. Overall, this architecture therefore has the potential to make existing verifiers information flow aware with substantially less effort than building a new tool from scratch.

Even though this approach has strong advantages, there are several open questions that must be addressed to make it useful and widely applicable:

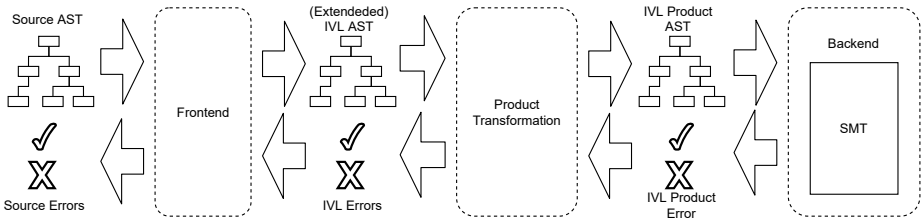

Fig. 1. Proposed architecture for information flow verifiers. The existing encoding from source to IVL (frontend) as well as the proof search (backend) can be reused. The product construction needs to support only the (relatively small) IVL and can be reused across different verifiers.

1. Soundness: Given an IVL encoding and a product construction that are individually sound, is the resulting combination always sound as well?
2. Concurrency: There is a substantial number of verifiers that verify concurrent source programs by encoding them into (sequential) IVLs. Can we soundly verify information flow security of concurrent programs based on the a product program of the sequential IVL encoding?
3. Performance: Product constructions cause a performance penalty for verification. Does this overhead prevent the construction of useful verification tools in practice?

In this paper, we answer these three questions. We focus our investigation on modular product programs [19], a kind of product program that allows modular verification and is well-suited for precise specification and verification of information flow security. We make the following contributions:

- We show that the combination of sound IVL encodings and sound product constructions can indeed be unsound in practically-relevant cases. We identify a novel condition on IVL encodings that ensures the soundness of the overall workflow. We show how to adjust existing unsound encodings on the example of a commonly-used encoding for dynamically-bound method calls (Sec. 3).
- We show for the common case of data race free programs using locks that it is possible to verify both possibilistic and probabilistic noninterference for concurrent programs using sequential product programs. Furthermore, we demonstrate that existing criteria for verifying information flow security are insufficient in this setting; we provide alternative criteria that are sound and show how to encode them in a product program (Sec. 4).
- We implement the approach for Nagini [18], an automated, modular verification tool for a large subset of Python, built on top of the Viper IVL [35]. We evaluate the performance impact of the product construction and show that, while worse than a custom-made information flow verifier, performance is acceptable for real-world use (Sec. 5). Our implementation and evaluation are available as an artifact [17].

These results demonstrate that the proposed approach can indeed be used to retrofit an existing verifier to *soundly* check information flow security, even for concurrent programs. The resulting tool, made with only a fraction of the effort required for the development of a new verifier, can compete with custom-made tools in its expressiveness at an acceptable performance cost.

2 Preliminaries

In this section, we introduce the necessary background about noninterference and product programs.

2.1 Noninterference

A common way of formalizing information flow security is *noninterference* [24]. Informally, noninterference specifies that the secret (or *high*) inputs of a program do not influence the values of its public (or *low*) outputs. We will not define a formal semantics here, but just assume that there is a steps-to relation $\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle$ that relates program configurations consisting of a store σ and a statement s.

We formalize noninterference as a property of pairs of program executions (that is, a *2-hyperproperty* [12]) as follows:

Definition 1. *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies noninterference iff for all σ_1, σ_2 and σ_1', σ_2', if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \sigma_1'\rangle$ and $\langle s, \sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2'\rangle$ then $\forall x \in O_l.\sigma_1'(x) = \sigma_2'(x)$.*

Note that in this definition (and throughout this paper unless stated otherwise), we do not consider non-terminating executions, i.e., we focus on verifying termination-insensitive noninterference.

2.2 Modular Product Programs

Proving hyperproperties requires reasoning about multiple (here, two) executions of a program. However, hyperproperties can be reduced to properties of a single execution by using *self-composition* [6] or *product programs* [4]. The idea is to duplicate a program's state space by creating two renamed copies of all variables, one for each execution (we write $x^{(i)}$ for the ith renaming of variable x, and lift this notation to expressions), and to transform each statement so that it has the effect of the original statement on both copies of the state. Unlike self-composition, which achieves this effect by simply duplicating every statement, *modular product programs* [19] do not duplicate loops and method calls, and instead encode differing control flow through *activation variables*, which represent, for each execution, whether or not it is active (i.e., it executes the code) at the current point in the program. This approach results in a structural alignment

```
def bar(z): ...

def foo(x):
  if x > 0:
    y = 1
  else:
    y = 2
  bar(y > 0)
```

```
1  def bar(p1, p2, z1, z2): ...
2
3  def foo(p1, p2, x1, x2):
4    p1t = p1 and x1 > 0
5    p2t = p2 and x2 > 0
6    p1e = p1 and not (x1 > 0)
7    p2e = p2 and not (x2 > 0)
8    if p1t: y1 = 1
9    if p2t: y2 = 1
10   if p1e: y1 = 2
11   if p2e: y2 = 2
12   if p1 or p2:
13     bar(p1, p2, y1 > 0, y2 > 0)
```

Fig. 2. A modular product program (on the right) of the program on the left.

of both program executions, which allows one to use method specifications and loop invariants that relate both executions, as we discuss below. We denote the product of statement s under activation variables $p1$ and $p2$ as $[\![s]\!]^{\mathring{p}}$.

Fig. 2 shows an example program and the respective product program. For both functions, the product program duplicates the parameters of the original function and adds boolean activation variables p1 and p2. Control structures like conditionals are encoded by creating a set of new activation variables (lines 4-7). For example, p1t represents whether the first execution is active in the then-branch of the conditional, which is the case if it was active at the beginning of the function and the if-condition is true for the first execution. Conversely, p2e represents whether the second execution is active in the else-branch. Primitive statements like assignments are then executed under the condition that their execution is active at the current point in the execution (lines 8-11). Crucially, the method call to bar is *not* duplicated; it is executed if at least one execution is active at the call site, and the values of the current activation variables are passed to the function, meaning that if an execution is inactive at the call site, no state changes will be performed for that execution in the called method.

Because a single method call in the product represents the calls from both executions, one can reason about method calls modularly in terms of *relational* specifications, i.e., specifications that relate behavior of two executions of the method, as opposed to *unary* specifications that describe only a single execution. Relational specifications are encoded as ordinary specifications in the product program that relate parameters from the two different executions.

As an example, assume that bar prints the value of its input z, which must therefore be low. We can express this as a (relational) precondition $low(\mathsf{z})$, which can be encoded as the precondition $\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{z1} = \mathsf{z2}$ in the product of bar.

Events the attacker can observe (such as I/O) must not happen depending on a secret, to avoid leaking secret data. It is, thus, useful to express in specifications that the control flow at the current program point is low, i.e., whether the current statement is executed does not depend on secret data. This property is denoted in specifications as *lowEvent*. We generally write $\lceil P \rceil^{\mathring{p}}$ for the encoding of assertion P under activation variables $p1$ and $p2$; $\lceil lowEvent \rceil^{\mathring{p}}$ is then defined as $\mathsf{p1} = \mathsf{p2}$.

A unary (that is, non-relational) predicate Q, such as a standard method pre- or postcondition, is encoded in the product program as applying to each active execution, i.e., $\lceil Q \rceil^{\tilde{p}}$ is defined as $\mathsf{p1} \Rightarrow Q^{(1)} \wedge \mathsf{p2} \Rightarrow Q^{(2)}$.

Compared to type systems and taint analyses, verification based on product programs allows for much more precise reasoning. Assume for example that foo's parameter x is high. Nonetheless, we can show that the example does not leak information, since the precondition of bar, *low*(z), will always be fulfilled ($\mathsf{y} > 0$ is true independently of the value of x). In contrast, security type systems would flag y as high, since it is assigned to under a high guard, leading to imprecision.

In addition to ordinary noninterference, modular product programs can also be used to encode more advanced security properties, including termination-sensitive noninterference, value-dependent sensitivity [36], and a form of declassification [19].

3 Sound Products of IVL Encodings

In this section, we address the first question from the introduction, namely, whether we can soundly combine an existing encoding into an IVL with a product construction. We first describe the proposed architecture in greater detail. Then we show a potential soundness issue and define a sufficient criterion on the IVL encoding for the entire approach to be sound. Finally, we discuss an example of a common encoding pattern that violates the criterion, show that it is indeed unsound, and propose an alternative sound encoding.

3.1 Proposed Architecture

The architecture proposed in the introduction (Fig. 1) enables the construction of information flow aware verifiers with relatively little effort, by reusing most of the frontend encoding of the source language to an IVL as well as the entire backend proof search. The only major change that is necessary is that the frontend and potentially the IVL have to be extended to allow for the use of information flow assertions in specifications. Crucially, the frontend does not have to know their meaning; it can treat relational source-level assertions like *low*(e) like ordinary unary predicates and simply translate them to their counterparts on the IVL level. IVL-level relational assertions will then be translated to ordinary assertions during the product transformation.

In the remainder of this paper, we will generally assume that the existing IVL encoding is used unchanged, and point out when changes need to be made.

3.2 Soundness Issue

Surprisingly, combining a sound encoding from source language to IVL with a sound IVL-level product construction may result in a verification technique that is *unsound* in the presence of relational specifications. Consider the source program in Fig. 3 (left), where P is some predicate.

```
def foo(x):
  if x > 7:
    y = 5
  else:
    y = 7
  assert P(y)
```

```
def foo(x):
  assert x > 7 ? P(5) : P(7)
```

Fig. 3. Example of an encoding that is unsound in our setting. The program on the left can be encoded into a conditional statement (identical to the source program, modulo language syntax) or to the program on the right; the latter leads to unsoundness if P is a relational predicate.

A frontend could encode the body of foo into an identical (modulo syntax) conditional statement on the IVL level (assuming the IVL provides conditionals, assignments, and assert statements). Alternatively, it could produce the encoding shown in Fig. 3 (right), which directly asserts a sufficient precondition of the source program. If P is a unary predicate, both encodings are sound: If they verify, the original program is correct. However, if P(y) is a relational predicate, for instance, *low*(y), then the encoding on the right is *unsound*: *low*(5) and *low*(7) are trivially true (since $5 = 5$ and $7 = 7$), so the assertion in the encoded program trivially passes, yet the original program is clearly incorrect: If x is greater than 7 in one execution but less in the other, y will have different values in both executions, and will therefore not be low.

The underlying reason is that the encoding on the right does not encode the exact behavior of the source program; it encodes a verification condition computed by the frontend that is sound if assertions are unary, but may not be sound for relational assertions.

We will now (1) formalize this intuition and derive a sufficient condition for the soundness of an encoding in this approach, and (2) show an example of this problem occurring in real frontends, and describe how it can be solved.

3.3 Soundness Criterion

We write Σ and S for states and statements of the source language, and σ and s for states and statements of the IVL. States may contain, for example, a mutable heap and a variable store. For simplicity, we assume that both source and IVL statements contain a statement `skip` that represents a finished computation. We also assume that there is a small-step transition relation $\rightarrow$ for both languages, and that the standard notion of Hoare triple validity $\vDash \{P\}s\{Q\}$ is defined for the IVL. We let P and Q range over (source and IVL level) assertions from a standard assertion language extended with *low*(e) and *lowEvent*, and assume a standard definition of assertion validity for pairs of states.

We define an encoding to be a triple $\langle \alpha, \cong, \beta \rangle$, where $\alpha : S \rightarrow s$ is an encoding from source statements to statements of the target language (i.e., the IVL), β similarly encodes assertions to the target language, and $\cong$ relates source language states to corresponding target language states.

We first define the desired relational soundness property, which expresses that if an encoded Hoare triple holds for the encoded program, then the original property holds for all pairs of executions of the source program:

Definition 2. $\langle \alpha, \cong, \beta \rangle$ *is* relationally sound *iff, for all* $S, \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2', P, Q$*, if* $\vDash \{\lceil \beta(P) \rceil^{\mathring{p}}\} [\![\alpha(S)]\!]^{\mathring{p}} \{\lceil \beta(Q) \rceil^{\mathring{p}}\}$ *and* $\Sigma_1, \Sigma_2 \vDash P$ *and* $\langle S, \Sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \Sigma_1' \rangle$ *and* $\langle S, \Sigma_2 \rangle \rightarrow^* \langle \texttt{skip}, \Sigma_2' \rangle$*, then* $\Sigma_1', \Sigma_2' \vDash Q$*.*

Product programs represent the operational behavior of two program executions by the operational behavior of a single product program execution. The unsoundness shown before is caused by the fact that the encoding into the IVL does not reflect the operational behavior of the conditional statement (replacing it by an assertion of a sufficient precondition) and, thus, the resulting product does not soundly reflect two executions of the source program.

We call an encoding that preserves the operational behavior of the source program *operational*: It encodes every step of the source program into some number of steps of the target program so that c initial states result in matching end states. Similarly, it encodes specifications from the source level into target-level specifications that hold in matching states. We can formalize this intuition by requiring that the source and target programs are connected by the simulation relation $\cong$:

Definition 3. $\langle \alpha, \cong, \beta \rangle$ *is an* operational *encoding if: (1) for all* $\Sigma, \Sigma', \sigma, S, S'$*, if* $\langle S, \Sigma \rangle \rightarrow \langle S', \Sigma' \rangle$ *and* $\Sigma \cong \sigma$*, then* $\langle \alpha(S), \sigma \rangle \rightarrow^* \langle \alpha(S'), \sigma' \rangle$ *for some* σ' *s.t.* $\Sigma' \cong \sigma'$*, and (2) if* $\Sigma \cong \sigma$ *then* $\Sigma \vDash P$ *iff* $\sigma \vDash \beta(P)$*.*

Note that this notion allows the encoding to overapproximate the behaviors of the source program, i.e., admit steps that are not possible on the source level, but not vice versa.

For the example in Fig. 3, it is easy to see that this criterion is fulfilled by the left encoding: the source and IVL programs are identical (modulo syntax), matching states are identical states (modulo state encodings), and the behavior of both programs is identical. The encoding on the right, however, is *not* operational: While the left program modifies the state, the right program never performs any state modification.

We now show that operationality is sufficient for relational soundness:

Theorem 1. *If* $\langle \alpha, \cong, \beta \rangle$ *is operational then it is relationally sound.*

Note that operationality is a sufficient but not necessary condition; encodings of verification conditions may be sound for relational verification as well. The main advantage of applying the operationality criterion instead of directly reasoning about relational soundness is that, since operationality represents the simple notion that the IVL program performs equivalent steps and equivalent state changes to the source program, it is intuitive and easy to check whether a given encoding is operational. Additionally, some encodings (like the one Vercors uses for parallel blocks) are not operational, but can be seen as simplified versions of a possible operational encoding that generate the same proof obligations; these can also be quickly identified as relationally sound.

3.4 Practical Relevance

In most existing frontends, the encoding of virtually all source language constructs is operational; the main appeal of IVLs is, after all, that frontends *do not* have to compute verification conditions, but can instead "compile" input programs into an IVL without worrying about the verification process itself. However, many frontends still use non-operational encodings at least for *some* language constructs. Examples for this are VCC's encoding of local blocks, Dafny's encoding of calls on traits, Prusti's encoding for loops, and Nagini's encoding of dynamically-bound calls, which we will discuss in detail in the next subsection. Additionally, as we will discuss in Sec. 4, all encodings of concurrent source languages into sequential IVLs necessarily have some non-operational elements.

Where non-operational encodings are used, this is often intentional to enable modular verification, since operational encodings for some language constructs are inherently non-modular (see the example in the next subsection). In practice, one can therefore use the operationality criterion to quickly check that the existing encoding is sound for the vast majority of source language statements, and subsequently check the few remaining ones for relational soundness in detail.

3.5 Example: Dynamically-Bound Calls

In this section, we show a real example of an unsound encoding of dynamically-bound calls that violates the operationality criterion, and show how to derive a sound alternative.

Statically-bound method calls, i.e., calls whose exact target is fixed at compile time, can be encoded as procedure calls on the IVL level, which yields an operational encoding if the operational semantics of the IVL treats calls analogously to the source semantics. The IVL verifier might later reason about calls in terms of pre- and postconditions instead of actually performing a call, but this transformation is not relevant here as long as the product program is constructed *before* such a desugaring step.

However, the same approach does not work for dynamically-bound calls, i.e., calls whose target is chosen at runtime based on the type of the call's receiver. Since the implementation to be executed is generally not known during modular verification, it is not possible to encode dynamically-bound calls as procedure calls with the usual operational semantics (and existing IVLs do not offer dynamically-bound calls). Therefore, dynamically-bound calls are typically (e.g., in Dafny and Nagini) directly encoded using the method specification. Additional, separate proof obligations enforce that all overrides of a method respect *behavioral subtyping* [34], i.e., live up to the specification of the overridden method.

Consider method `A.foo` in Fig. 4 (left), which returns a constant integer and guarantees in its postcondition that the result is low. A dynamically-bound call `a.foo()`, where `a` has the static type `A`, will be encoded as an assertion of the (here, trivial) precondition of `A.foo`, followed by an assumption of the postcondition (we ignore side effects here for simplicity).

```
class A:
  def foo(self) -> int:
    # ensures low(result)
    return 0
```

```
class B(A):
  def foo(self) -> int:
    # ensures low(result)
    return 1
```

Fig. 4. Example of a problematic method override. B.foo overrides A.foo and has a compatible specification, but the implementations return different values.

This encoding is sound if foo has a purely unary specification, without any relational parts. However, it does *not* fulfill our operationality criterion: The semantics of the source program performs a call to an implementation of foo (selected based on the dynamic type of a), whereas the IVL encoding directly encodes the proof obligations (similarly to the example from Fig. 3).

Since the encoding is not operational, we have to check whether it is still relationally sound. Method B.foo in Fig. 4 (right), which overrides A.foo, shows that it is not. B.foo's contract is identical with that of A.foo, so behavioral subtyping holds trivially. B.foo's implementation satisfies the contract because it also returns a constant (but, importantly, a different one). Now, if a client calls a.foo() and, depending on a secret, the dynamic type of a is either A or B, then, depending on the secret, the result will be either 0 or 1. With the standard encoding of dynamically-bound calls outlined above, however, the client will assume the postcondition of A.foo and will therefore incorrectly conclude that the returned result is low.

To avoid this unsoundness while retaining the ability to use relational specifications[1], the problematic encoding must be replaced, either with an operational one, or with a different non-operational encoding that is sound for relational specifications. The former option is not applicable here: An operational encoding for dynamically-bound calls would essentially have to case split on the dynamic type of the receiver and invoke the appropriate override. Since such an encoding is inherently non-modular (all possible overrides need to be known), we follow the alternative option: we give an example of a non-operational, but sound encoding.

For our new encoding we exploit the fact that the standard encoding is unsound only if the two executions of the program resolve the dynamically-bound call to two different implementations, that is, if the dynamic types of the receiver differ in the two executions. We reflect this observation by adjusting the encoding of pre- and postconditions as follows: (1) If the postcondition of a method guarantees that an expression is low, we assume this at the call site *only if* the dynamic type of the receiver is also low, that is, the calls in the two program executions are resolved to the same implementation. (2) Similarly, if a precondition requires that the call is a low event, we enforce that the receiver type is low in addition to the usual criterion for low events. Low events typically perform observable behavior such as I/O; it is therefore important that the *same*

[1] One could, of course, forbid the use of relational specifications in some places to trivially avoid the unsoundness; this, however, is typically not useful in practice.

observable behavior is produced, independent of the receiver type. The meaning of low-assertions in preconditions remains unchanged, because the requirement of a method to receive low arguments is independent of the invoked implementation and must, thus, not be weakened. *lowEvent*-assertions are generally not allowed in postconditions, where they add no expressiveness.

We encode this adjustment as follows:

$$\lceil low(e) \rceil^{\mathring{p}}_{post_r} = (p^{(1)} = p^{(2)} \wedge type(r^{(1)}) = type(r^{(2)})) \Rightarrow e^{(1)} = e^{(2)}$$
$$\lceil lowEvent \rceil^{\mathring{p}}_{pre_r} = p^{(1)} = p^{(2)} \wedge type(r^{(1)}) = type(r^{(2)})$$

where $type(e)$ represents the dynamic type of expression e, $\lceil P \rceil^{\mathring{p}}_{post_r}$ is the encoding of P in the postcondition of a call with receiver r, and $\lceil P \rceil^{\mathring{p}}_{pre_r}$ represents the same for the precondition. We leave the remaining encoding untouched, meaning that we can summarize the resulting encoding as follows:

1. We keep the existing check for behavioral subtyping for all overrides; this prevents, for example, that A.foo is overridden with a method that simply returns a secret value and therefore leaks information into the result.
2. We keep the existing encoding of dynamically-bound calls as an assert followed by an assume, but interpret $low(e)$ in preconditions and *lowEvent* in postconditions as shown above.

In the example above, this encoding lets the caller assume that the result is low only if it can prove that the dynamic type of a is low.

The adjusted encoding is indeed sound:

Theorem 2. *Let S_c be of the form $x{:=}r.m()$, where r has static type A, and let $pre_{A.m}$ and $post_{A.m}$ be the pre- and postcondition of $A.m$. Assume that the implementation of A.m and its overrides fulfill their specifications and satisfy behavioral subtyping. Then the described encoding of S_c is relationally sound.*

Note that this encoding is incomplete, since it is not aware that two different receiver types can lead to the same implementation being called (e.g., if one type inherits from the second and does not override the called method). Alternative encodings could explicitly represent this possibility. Conversely, one could approximate further (while remaining sound) by requiring the receiver *values* to be low, not just their types, in encodings that do not model dynamic types.

4 Product Programs and Concurrency

Automated verification of information flow security for concurrent programs is challenging because one needs to reason about a pair of executions that may have different thread interleavings. In fact, we are aware of only one tool that currently allows this, SecC, which automates SecCSL, a concurrent separation logic for information flow security proofs [20]. A product construction applied directly to concurrent programs would have to faithfully represent all combinations of

potential thread interleavings, which makes verification infeasible. Consequently, to the best of our knowledge, no such product construction exists.

For trace properties, many existing verifiers avoid reasoning about all possible thread interleaving by employing a program logic (such as concurrent separation logic [38]) that essentially reduces verification to sequential reasoning and allows concurrent verification problems to be encoded into sequential IVLs. Examples for such verifiers include Vercors and Nagini (using the Viper IVL), as well as Chalice [33], VCC, and Spec# (using the Boogie IVL).

In this section, we show how to use IVL-level product programs to extend such verifiers to handle information flow. We first describe how existing IVL encodings for concurrent languages work, and subsequently show how we can use similar principles to apply an IVL-based product construction, and which additional proof obligations we must fulfill to ensure that no flows exist as a result of concurrency. We will do this for two different notions of information flow security for concurrent programs, *possibilistic* and *probabilistic* noninterference; however, the principles behind the approach may also extend to alternative notions of information flow security such as observational determinism [49].

Our goal is to describe a technique that applies to a wide range of source languages, IVLs, proof techniques, and encodings. Therefore, we focus on the high-level concepts, instead of formalizing them for one specific setting.

4.1 Concurrent IVL Encodings

Since all IVLs we are aware of are sequential languages, encodings from concurrent source languages to IVLs do not model the exact behavior of the original language, in particular, the aforementioned thread interleavings (i.e., these encodings are non-operational). Instead, they encode a verification condition that ensures that the original program is correct for *every possible* thread interleaving.

While the exact proof techniques differ between frontends, and can be based for example on Concurrent Separation Logic (CSL) [38] or ownership [13,25,27], they generally follow a common pattern [32]: They prove that the source program is data race free, which ensures that thread interactions need to be considered *only* at well-defined synchronization points, for instance, upon acquiring or releasing a lock. The code between such interaction points can be considered to execute without interference from other threads, and thus can be reasoned about as if it were sequential.

We focus on locks here, but other synchronization primitives are handled analogously. Program logics based on CSL or ownership systems formally connect a lock and the heap locations it protects, such that these locations may be accessed only while holding the respective lock. In addition, they associate locks with an invariant that constrains the values of the heap locations it protects. When acquiring a lock, a thread may assume that this lock invariant holds, and when releasing a lock, it has to prove that the invariant is reestablished. A frontend can encode this into an IVL as depicted in Fig. 5.

Our solution for information flow verification in concurrent programs follows the same basic approach: We exploit that code between lock operations can

$$
\begin{aligned}
enc(\mathsf{l}.\,\mathsf{acquire}()) &= \textit{// gain access to protected memory};\\
&\quad\ \texttt{assume}\ \mathit{Inv}(\mathsf{l})\\
enc(\mathsf{l}.\,\mathsf{release}()) &= \texttt{assert}\ \mathit{Inv}(\mathsf{l});\\
&\quad\ \textit{// lose access to protected memory}
\end{aligned}
$$

Fig. 5. Standard IVL encoding of lock operations. $\mathit{Inv}(\mathsf{l})$ denotes the invariant constraining the memory protected by lock l.

be considered to execute without interference, and that we can therefore use ordinary sequential product programs to reason about this code. To capture the thread interactions at synchronization points, we extend lock invariants to contain relational assertions (which can prescribe that some values protected by the lock are low), and add additional checks around lock operations to ensure that they do not give rise to unwanted information flow.

4.2 Possibilistic Noninterference

For concurrent programs, standard noninterference is too strict a property because concurrent programs are usually non-deterministic. One way of approaching this problem is to instead verify *possibilistic noninterference*, which enforces that high information does not influence the *possible* values of low outputs, i.e., if some combination of low output values is reachable from an initial state, then the same combination of low output values must still be reachable using *some* possible thread schedule after arbitrarily changing the high inputs. Possibilistic noninterference can be defined as follows:

Definition 4. *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies possibilistic noninterference iff for all σ_1, σ_2 and σ_1', if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \to^* \langle \texttt{skip}, \sigma_1'\rangle$ then $\langle s, \sigma_2\rangle \to^* \langle \texttt{skip}, \sigma_2'\rangle$ for some σ_2' s.t. $\forall x \in O_l.\sigma_1'(x) = \sigma_2'(x)$.*

Note that this property allows high inputs to influence the *probability* of different outputs and may therefore not be desirable in all scenarios; we discuss a stronger notion of noninterference in the next subsection.

Since we build on a proof technique that ensures data race freedom, we can see each program trace as a sequence of local operations and lock operations by specific threads, where (1) every local operation depends only on previous (local or lock) operations of the same thread, and (2) every lock operation depends only on the previous local operations of the same thread and all previous lock operations (of arbitrary threads). As a result, we can (akin to partial order reduction) rearrange segments freely as long as we retain the overall order of lock operations and the order of operations of every specific thread; in particular, we can rearrange a trace so that it consists of a number of *segments*, such that in each segment, one thread executes any number of local operations and then one lock operation.

$$
\begin{aligned}
\mathit{poss}(\mathsf{l}\,.\,\mathsf{acquire}\,()) &= \texttt{assert}\ \mathit{lowEvent};\\
&\quad\ \texttt{assert}\ \mathit{low}(l);\\
&\quad\ \texttt{assume}\ \mathit{Inv}(\mathsf{l})\\
\mathit{poss}(\mathsf{l}\,.\,\mathsf{release}\,()) &= \texttt{assert}\ \mathit{lowEvent};\\
&\quad\ \texttt{assert}\ \mathit{low}(l);\\
&\quad\ \texttt{assert}\ \mathit{Inv}(\mathsf{l})\\
\mathit{poss}(\texttt{while}\ (e)\ \texttt{do}\ \{s\}) &= \texttt{assert}\ \mathit{low}(e);\\
&\quad\ \texttt{while}\ (e)\ \texttt{do}\ \{\mathit{poss}(s); \texttt{assert}\ \mathit{low}(e)\}
\end{aligned}
$$

Fig. 6. Statement encoding for possibilistic information flow security. For loops, we check that the loop guard is low, ensuring that termination is also low.

Based on this observation, we impose proof obligations that ensure the following property: For every program trace with some schedule and some low and high inputs, and for arbitrary different high inputs, there exists a second trace such that: (1) Both traces include the same lock operations performed by the same threads, in the same order, and (2) at each lock operation, the lock's invariant holds; in particular, the relational assertions of the lock invariant correctly relate the state protected by the lock in both traces.

To enforce this property, we devise four proof obligations that can be checked thread-locally:

1. Every lock operation o is a low event, i.e., if a thread executes o in the first execution, it will also execute o in the second execution.
2. Termination of the local code *before* the lock operation does not depend on secret data; i.e., if lock operation o is reached in the first trace, it will also be reached in the second trace.
3. o operates on the same lock in both executions, i.e., the lock is low.
4. If o releases the lock, i.e., makes a new lock state public, this lock state fulfills the relational invariant, meaning that heap operations meant to be low are indentical in both executions after the lock operation.

Note that, even though the lock operations of both traces are closely aligned, their local operations may differ. For instance, a thread may branch on a high guard as long as no lock operation is performed before the control flow re-joins.

The above checks are sufficient to satisfy Def. 4. The proof goes by induction on the number of segments of the traces and leverages the soundness of sequential verification within each segment.

Encoding. The aforementioned checks can simply be checked as part of the encoding of lock operations. We adjust the encoding from Fig. 5 for possibilistic noninterference as shown in Fig. 6. For thread acquire and release, the assertions of *lowEvent* and $\mathit{low}(l)$ directly ensure properties (1) and (3). Assuming and asserting the lock invariant works as in the standard IVL encoding for concurrent programs, but now this invariant can be relational, ensuring property (4). The

```
def main(secret: bool) -> None:
    c = Cell()
    l = CellLock(c)
    l.acquire()
    c.val = 4
    if secret:
        l.release()
        l.acquire()
    c.val = 5
    c.release()
```

Fig. 7. Possibilistic information flow violation via a secret-dependent lock release. The Cell state 4 is visible to other threads only if secret is True.

condition on while loops is used to ensure property (2), which can be done simply by asserting that the loop condition is low for every loop in the program (we assume, for simplicity, that there is no infinite recursion).

Discussion. With our verification technique, the product construction on the IVL level does not need to be aware of concurrency in *any* way; applying the standard sequential product construction to the updated encoding is sufficient to ensure possibilistic noninterference in concurrent programs.

To the best of our knowledge, we are the first to consider possibilistic information flow in a setting with locks, and therefore the first to propose that the order of lock operations must be constrained. The example in Fig. 7 demonstrates that this requirement is indeed necessary to prevent unwanted information flow: The CellLock protects the val field of a Cell object, which is intended to be low. The code unconditionally sets the field to two constants (first to 4, then to 5), which should be allowed since the constants are low. However, whether the lock is *released* while the cell has value 4 depends on a secret. As a result, when a different thread acquires the lock and sees that the value is 4, this leaks that the secret *must* have been true.

Another example that illustrates the requirement to ensure that high data does not influence *which* lock a lock operation accesses can be found in Fig. 8. Here, two locks are created, and thread 1 acquires the first one. Thread 2 acquires, depending on the secret, either the same lock or a different one. This influences the possible results of the program: If both threads acquire the same lock, then the **print** statements of one thread cannot be interleaved with those of the other, otherwise they can. As a result, if the attacker observes the pattern 1212 (or any other interleaving of 1s and 2s), they know with certainty that the two threads acquired different locks and secret must therefore be False.

The necessity to prevent termination differences in a concurrent setting has been recognized before in work on security type systems [45].

```
def thread1(l: Lock) -> None:
    # requires lowEvent
    l.acquire()
    print(1)
    print(1)
    l.release()

def thread2(l: Lock) -> None:
    # requires lowEvent
    l.acquire()
    print(2)
    print(2)
    l.release()

def main(secret: bool) -> None:
    l1 = Lock()
    l2 = Lock()
    if secret:
        l = l1
    else:
        l = l2
    fork thread1(l1)
    fork thread2(l)
```

Fig. 8. Possibilistic information flow violation through locks. If secret is true, both threads acquire the same lock, and their critical sections cannot be interleaved.

```
def thread1(l: Lock, c: Cell):
    ctr = 0
    for i in range(100):
        ctr += 1
    l.acquire()
    c.val = 1
    l.release()

def thread2(l: Lock, c: Cell, secret: int):
    ctr = 0
    for i in range(secret):
        ctr += 1
    l.acquire()
    c.val = 2
    l.release()
```

Fig. 9. Example of probabilistic information flow. With a non-deterministic scheduler, secret does not influence the set of possible outputs, but a greater secret leads to higher probability of seeing a final cell value of 2.

4.3 Probabilistic Noninterference

Possibilistic noninterference is too imprecise for many applications. Fig. 9 illustrates the problem: The final value of c.val can be either 1 or 2, that is, possibilistic noninterference holds. However, with most schedulers, a final value of 2 is much more likely for greater secret values than for lower values because the assignment of 1 is more likely to happen before the assignment of 2.

A stronger notion of noninterference that forbids such leaks is *probabilistic* noninterference, which requires that two executions from low-equivalent initial states will produce the same low outputs with the same probabilities.

Definition 5. *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies probabilistic noninterference iff for all σ_1, σ_2 and σ_1', if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \sigma_1'\rangle$ with probability p then $\langle s, \sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2'\rangle$ with probability p for some σ_2' s.t. $\forall x \in O_l.\sigma_1'(x) = \sigma_2'(x)$.*

The information flow in Fig. 9 is caused by secret data influencing the timing of thread 2, which in turn may affect the relative order of modifications of shared variables. To prevent secrets from influencing the timing of operations, we additionally assert that every branch condition in the program is low, meaning that the two executions will always follow the same code path, which leads to the adjusted encoding in Fig. 10. Note that the check that branch conditions are

$$
\begin{aligned}
&prob(\mathsf{l}.\,\mathsf{acquire}\,()) && = \texttt{assert}\ low(l);\\
&&&\quad \texttt{assume}\ Inv(\mathsf{l})\\
&prob(\mathsf{l}.\,\mathsf{release}\,()) && = \texttt{assert}\ low(l);\\
&&&\quad \texttt{assert}\ Inv(\mathsf{l})\\
&prob(\texttt{while}\ (e)\ \texttt{do}\ \{s\}) && = \texttt{assert}\ low(e);\\
&&&\quad \texttt{while}\ (e)\ \texttt{do}\ \{prob(s); \texttt{assert}\ low(e)\}\\
&prob(\texttt{if}\ (e)\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}) && = \texttt{assert}\ low(e);\\
&&&\quad \texttt{if}\ (e)\ \texttt{then}\ \{prob(s_1)\}\ \texttt{else}\ \{prob(s_2)\}\\
&prob(r.m()) && = \texttt{assert}\ low(type(r));\\
&&&\quad r.m()
\end{aligned}
$$

Fig. 10. Statement encoding for probabilistic information flow security.

low must also be performed for any implicit branches; e.g., with the encoding of dynamically-bound calls shown before, we must now assert that the type of the receiver of every such call is low. Also note that since we enforce that branches are low, the *lowEvent* conditions we showed in the possibilistic encoding will be trivially fulfilled and can be omitted here. However, we still need to assert that acquired and released lock references are low. This last requirement has not been discussed in previous work (whereas forbidding high branches is standard practice in type systems and program logics [36]).

With this adjusted encoding, probabilistic noninterference can be verified using simple assertions in the IVL encoding and subsequently performing a standard product construction on the IVL level. So, in summary, this approach lets us extend existing verifiers for concurrent programs to verify both possibilistic and probabilistic noninterference with very small changes in the frontend, and without requiring any changes on the level of the IVL (except the ability to write relational specifications) and the product construction.

5 Implementation and Evaluation

In this section, we evaluate the performance of the proposed architecture, by extending the previously information flow unaware Nagini verifier for Python [18] according to our design. We will first briefly describe Nagini and the adaptations we needed to make, then evaluate the performance overhead generated by the product transformation, and subsequently evaluate the implementation on a number of information flow examples, comparing it to SecC [20] in the process.

5.1 Nagini

Nagini is an automated verifier for statically-typed Python 3 programs. It supports a large subset of the Python language, comprising features like exception handling, polymorphism, dynamic field creation, and concurrency. Reasoning about some of these features is quite intricate even without the overhead of a

product construction, so we believe that Nagini is a good target to evaluate the performance of the proposed architecture for verifiers for complex languages.

Nagini encodes Python programs and their specifications into the Viper IVL [35], and then uses Viper's backend verifiers to automatically verify those programs using the Z3 SMT solver [15]. For concurrent programs, Nagini uses an encoding similar to the one described in Sec. 4, using implicit dynamic frames [44] (a flavor of separation logic [38, 40]) to prove data race freedom; as a result, we could modify its existing encoding as shown in Sec. 4 to prove both possibilistic and probabilistic noninterference for concurrent programs. Nagini's existing encoding from Python to Viper is almost entirely operational, we only adapted the encoding of dynamically-bound calls as shown in Sec. 3.5.

We extended Nagini's existing specification language to include information flow specifications and implemented the modular product program transformation for 2-hyperproperties for the existing Viper AST (enriched, again, with new AST nodes for information flow specifications). For convenience, we also slightly extended the Viper-based product transformation to directly transform statements that Nagini previously encoded using gotos, such as **break** and **continue** statements. The Viper extension for product programs[2] and the extended version of Nagini[3] are open source and available online.

5.2 Performance Overhead of the Product Construction

Our first goal is to evaluate the performance overhead generated by the product construction. We compared the verification times of Nagini's entire functional test suite with and without the product transformation enabled. The test cases range from small programs targeting specific language or specification constructs, to realistic code examples taken from programming tutorials. We ran each test five times on a warmed up JVM with the information flow extension enabled and disabled, without adding any information flow specifications. Our test system was a 12 core AMD Ryzen 3900X with 32GB of RAM running Ubuntu 20.04.1.

All tests report the same results with and without the product transformation, meaning that completeness is not impacted by the extension, and that we can indeed still reason about the entire language subset supported by Nagini. Without the product transformation, each test case takes between 3 and 9 seconds, with the majority taking between 3 and 5. For most cases, enabling the product construction leads to an increase in verification time that is clearly acceptable (less than 11% for half the tests, less than 30% for three quarters, and less than 100% for 90% of the tests). For five test cases, the slowdown is a factor between 5 and 12, and a single outlier (a quicksort implementation) has a slowdown factor of 17.5 and a resulting verification time of two minutes. We believe that the main reason for the large slowdown for these particular test cases is the use of quantifiers in their specifications (e.g., to specify properties of all elements in a list). Quantifier handling is difficult for automated verification in

[2] https://github.com/viperproject/silver-sif-extension
[3] https://github.com/marcoeilers/nagini

	LOC	Ann.	Prop.	T
banerjee	77	21	NI	5.19
constanzo	21	12	NI	5.39
darvas	38	18	NI	4.20
example	27	12	NI	5.39
Example-decl	27	12	NI	5.76
Example-term	8	4	TNI	3.59
joana-1-tl	22	7	NI	3.87
joana-2-bl	13	5	NI	3.64
joana-2-t	12	4	NI	3.72
joana-3-bl	36	15	TNI	3.55
joana-3-br	33	14	TNI	4.60
joana-3-tl	23	9	TNI	4.50
joana-3-tr	25	10	TNI	4.19
joana-13-l	11	2	NI	4.54
kusters	28	12	NI	4.35
naumann	27	17	NI	8.46
product	39	18	NI	11.35
smith	39	21	NI	6.81
terauchi1	10	3	NI	3.59
terauchi3	19	6	NI	3.69
terauchi4	18	8	NI	3.97
Fig. 4	19	6	NI	3.82
loop leak [45]	53	17	PS	4.92
high loop	24	11	PS	4.19
Fig. 7	23	8	PS	4.37
Fig. 8	36	15	PS	4.40
Fig. 9	34	15	PS	4.57

Table 1. Programs evaluated for proving information flow security. We show the total lines of code (LOC) including implentation and specification but excluding whitespace, lines of specification and proof annotation (Ann.), the property we proved (Prop., where NI = noninterference, TNI = termination sensitive noninterference, PS = possibilistic noninterference) and the verification time in seconds (T), averaged over five runs.

general, because unbounded chains of quantifier instantiations can occur during the proof search [16], and this problem seems to be exacerbated when using the product encoding.

We conclude that the performance impact of the product transformation is acceptable for most examples, but can be significant for programs with complex functional specifications.

5.3 Expressiveness and Comparison with SecC

In a second step, we evaluated the expressiveness and performance of our implementation on a number of challenging examples from the literature. In particular, we use the examples from the original paper about modular product programs [19] (sequential examples collected from various previous papers, translated to Python) and from this paper, both shown in Table 1, as well as examples taken from SecC [20], the only other automated verification tool for concurrent programs we are aware of, shown in Table 2. The latter table includes the CDDC case study [36], which models an embedded device that interacts simultaneously with multiple users and classified networks. Our examples represent the state of the art in automated information flow verification, requiring semantic reasoning that would not be possible in a type system, and using complex information flow specifications including declassification, termination-sensitive noninterference, and value-dependent sensitivity [36]. As mentioned before, these features can be easily encoded into modular product programs using existing techniques [19].

Nagini was able to verify all examples, which demonstrates that our approach can handle concurrent implementations and express complex noninterference properties. For the examples from Table 1, Nagini takes only between 3 and 12 seconds each. As for the tests from SecC, Nagini takes around five seconds

	LOC_N	Ann_N	LOC_S	Ann_S	Prop.	T_S	T_N	T_{NP}
SecC CAV	40	13	50	11	PR	1.33	4.21	3.56
SecC CDDC	278	105	214	47	PR	21.20	52.20	8.60
SecC CT	64	35	211	159	PR	1.87	5.41	3.97
SecC DB	100	48	256	167	NI	2.75	182.60	6.23
SecC Encrypt	29	12	49	18	NI	1.45	4.76	3.66

Table 2. Comparison with SecC. We show the total lines of code and lines of specification for Nagini (LOC_N, Ann_N) and SecC (LOC_S, Ann_S), the property we proved (Prop., where NI = noninterference, PR = probabilistic noninterference) and the verification time in seconds in both tools (T_N and T_S) and in Nagini *without* the product construction (T_{NP}), averaged over five runs.

for three of them, 52 seconds for the CDDC case study, and 183 seconds for an example involving a large number of quantifiers. We believe that 52 seconds for a complex case study is still acceptable, whereas the slowest example demonstrates that extensive use of quantifiers will lead to problematic performance in practice.

Table 2 shows that SecC is much faster than our implementation. However, SecC was designed and implemented for information flow verification from scratch, without being able to reuse code from an existing verifier, whereas our extended Nagini implementation could be implemented with minimal effort. Besides this crucial difference, Nagini and SecC differ in many other ways, e.g., in their supported language features, automation (see the table for required annotations), and specification styles. As a result, direct performance comparisons between the two are difficult; in fact, the unmodified version of Nagini *without* the product construction already takes more time than SecC on four out of five examples, likely as a result of the overhead required for modeling more complex language features.

6 Related Work

There are various existing type systems (e.g. [37,45]) and static analyses (e.g. [11, 23]) for proving information flow security. Compared to verification based on product programs, these are more automated, but less precise. Moreover, there are dedicated program logics for information flow verification, such as SecCSL [20], Covern [36], and Veronica [42], all of which allow proving probabilistic noninterference for concurrent programs based on different reasoning techniques. The implementation of the former in SecC is the only existing tool that automates information flow verification for concurrent programs, see Sec. 5.

Relational logics, such as Relational Hoare Logic [7] and Cartesian Hoare Logic [46], allow proving general relational program properties, which includes noninterference. However, while they tackle a more general problem, they generally work only for sequential programs. Some tools automate information flow verification using self-composition, e.g., for C [9] and for Java [41]. Compared to modular product programs, this approach generally does not allow for modular proofs of information flow security [19, 48].

Modular product programs were presented by Eilers et al. [19]. Other forms of product programs differ in the way executions are interleaved. While some keep executions in lock step [4], like modular product programs, others do not describe a deterministic product construction and allow for arbitrary interleavings [5]. In particular, Shemer et al. [43] propose property-directed self-composition, which dynamically determines how to compose and interleave different executions based on the property to be verified. Similarly, Strichman and Veitsman [47] propose a product-like construction that interleaves recursive functions whose executions are not in lock step. Recently, Pick et al. [39] showed how to automatically infer information flow specifications on modular product programs, which can likely be combined with the approach examined in this paper.

To the best of our knowledge, SymDiff [28] for the Boogie IVL is the only existing tool that constructs product programs on an IVL-level. SymDiff is a tool for differential program verification, which requires reasoning about pairs of executions of two different (but related) programs and is thus similar to hyperproperty verification; in fact, SymDiff has also been used to verify noninterference in the past [1]. The authors of SymDiff have proposed different techniques for modularly proving mutual function summaries, similar to relational specifications, one of which uses a kind of product construction [26,29]. However, they do not examine potential soundness problems arising from this approach, nor do they discuss if it can be applied to concurrent source programs.

7 Conclusion

We presented an approach for retrofitting existing IVL-based program verifiers to check information flow security using product programs. This approach allows reusing existing frontends to reduce the required implementation effort. We have shown when this technique is sound, that it can incorporate concurrency, and that it can be implemented in an existing verifier with acceptable performance.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70. USENIX Association, 2016.
2. V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, 2019.
3. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.
5. G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *LFCS*, volume 7734 of *LNCS*, pages 29–43. Springer, 2013.

6. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25. ACM, 2004.
8. A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132. ACM, 2012.
9. L. Blatter, N. Kosmatov, P. L. Gall, V. Prevosto, and G. Petiot. Static and dynamic verification of relational properties on self-composed C code. In *TAP@STAF*, volume 10889 of *LNCS*, pages 44–62. Springer, 2018.
10. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
11. Z. Chen, L. Chen, and B. Xu. Hybrid information flow analysis for python bytecode. In *IEEE WISA*, pages 95–100. IEEE Computer Society, 2014.
12. M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.
13. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.
14. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In *SEFM*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
15. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
16. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
17. M. Eilers, S. Meier, and P. Müller. Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security (Artifact), Apr. 2021.
18. M. Eilers and P. Müller. Nagini: A static verifier for Python. In *CAV (1)*, volume 10981 of *LNCS*, pages 596–603. Springer, 2018.
19. M. Eilers, P. Müller, and S. Hitz. Modular product programs. In *ESOP*, volume 10801 of *LNCS*, pages 502–529. Springer, 2018.
20. G. Ernst and T. Murray. SecCSL: Security concurrent separation logic. In *CAV (2)*, volume 11562 of *LNCS*, pages 208–230. Springer, 2019.
21. J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
22. J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
23. D. Giffhorn and G. Snelting. A new algorithm for low-deterministic security. *Int. J. Inf. Sec.*, 14(3):263–287, 2015.
24. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.
25. É. Goubault, J. Ledent, and S. Mimram. Concurrent specifications beyond linearizability. In *OPODIS*, volume 125 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
26. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *CADE*, volume 7898 of *LNCS*, pages 282–299. Springer, 2013.
27. B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM*, pages 137–147. IEEE Computer Society, 2005.

28. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV*, volume 7358 of *LNCS*, pages 712–717. Springer, 2012.
29. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *ESEC/SIGSOFT FSE*, pages 345–355. ACM, 2013.
30. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
31. K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *LASER Summer School*, volume 6029 of *LNCS*, pages 91–139. Springer, 2008.
32. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.
33. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
34. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
35. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.
36. T. C. Murray, R. Sison, and K. Engelhardt. COVERN: A logic for compositional verification of information flow control. In *EuroS&P*, pages 16–30. IEEE, 2018.
37. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at http://www.cs.cornell.edu/jif*, 2006.
38. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
39. L. Pick, G. Fedyukovich, and A. Gupta. Automating modular verification of secure information flow. In *FMCAD*, pages 158–168. IEEE, 2020.
40. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
41. C. Scheben and P. H. Schmitt. Verification of information flow properties of Java programs without approximations. In *FoVeOOS*, volume 7421 of *LNCS*, pages 232–249. Springer, 2011.
42. D. Schoepe, T. Murray, and A. Sabelfeld. VERONICA: expressive and precise concurrent information flow security. In *CSF*, pages 79–94. IEEE, 2020.
43. R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel. Property directed self composition. In *CAV (1)*, volume 11561 of *LNCS*, pages 161–179. Springer, 2019.
44. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.
45. G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. 2007.
46. M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, pages 57–69. ACM, 2016.
47. O. Strichman and M. Veitsman. Regression verification for unbalanced recursive functions. In *FM*, volume 9995 of *LNCS*, pages 645–658, 2016.
48. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
49. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, page 29. IEEE Computer Society, 2003.