- TOBIAS RUNGE, TU Braunschweig, Karlsruhe Institute of Technology, Germany
- MARCO SERVETTO, Victoria University of Wellington, New Zealand
- ALEX POTANIN, Australian National University, Australia
- INA SCHAEFER, TU Braunschweig, Karlsruhe Institute of Technology, Germany

9 Security-critical software applications contain confidential information which has to be protected from 10 leaking to unauthorized systems. With language-based techniques, the confidentiality of applications can 11 be enforced. Such techniques are for example type systems that enforce an information flow policy through 12 typing rules. The precision of such type systems, especially in object-oriented languages, is an area of active research: an appropriate system should not reject too many secure programs while soundly preserving 13 noninterference. In this work, we introduce the language SIFO<sup>1</sup> which supports information flow control for 14 an object-oriented language with type modifiers. Type modifiers increase the precision of the type system by 15 utilizing immutability and uniqueness properties of objects for the detection of information leaks. We present 16 SIFO informally by using examples to demonstrate the applicability of the language, formalize the type system, 17 prove noninterference, implement SIFO as a pluggable type system in the programming language L42, and 18 evaluate it with a feasibility study and a benchmark. 19

## $_{20}$ CCS Concepts: • Security and privacy $\rightarrow$ Information flow control.

Additional Key Words and Phrases: security, information flow, type system, mutation control, confidentiality, integrity

## 24 ACM Reference Format:

1 2

3 4

5

6

7

8

21

22

23

25

26

27 28

29

30

31

32

33

34

35 36

37

Tobias Runge, Marco Servetto, Alex Potanin, and Ina Schaefer. 2022. Immutability and Encapsulation for Sound OO Information Flow Control. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2022), 36 pages. https://doi.org/10.1145/XXX

## 1 INTRODUCTION

In security-critical software development, it is important to guarantee the confidentiality and integrity of the data. For example, in a client-server application, the client has a lower privilege than the server. If the client reads information from the server in an uncontrolled manner, we may have a violation of confidentiality; this causes the client to release too much information to the user. On the other hand, if the server reads information from the client in an uncontrolled manner, we may have a violation of integrity; this causes the server to accept input that has not been validated.

48 https://
 49

<sup>&</sup>lt;sup>1</sup>SIFO is an acronym for Secure Information Flow in an Object-oriented language

Authors' addresses: Tobias Runge, tobias.runge@kit.edu, TU Braunschweig, Karlsruhe Institute of Technology, Germany;
 Marco Servetto, marco.servetto@ecs.vuw.ac.nz, Victoria University of Wellington, New Zealand; Alex Potanin, alex.potanin@
 anu.edu.au, Australian National University, Australia; Ina Schaefer, ina.schaefer@kit.edu, TU Braunschweig, Karlsruhe
 Institute of Technology, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee
 provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and
 the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored.
 Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires
 prior specific permission and/or a fee. Request permissions from permissions@acm.org.

 <sup>&</sup>lt;sup>46</sup> © 2022 Association for Computing Machinery.

<sup>© 2022</sup> Association for Computing Machinery.

 <sup>&</sup>lt;sup>7</sup> 0164-0925/2022/1-ART1 \$15.00
 https://doi.org/10.1145/XXX

Language-based techniques such as type systems are used to ensure specific information flow policies for confidentiality or integrity [Sabelfeld and Myers 2003]. A type system assigns an explicit security type to every variable and expression, and typing rules prescribe the allowed information flow in the program and reject programs violating the security policy. For example, we can define a security policy as a lattice of security levels with the highest level high and the lowest level low, and an information flow from high to low is prohibited.

For simple while-languages, type systems to control the information flow are widely stud-56 ied [Hunt and Sands 2006; Li and Zhang 2017; Volpano et al. 1996]. We focus on the less researched 57 area of information flow control for object-oriented languages. Analysis techniques such as fine-58 grained taint analysis [Arzt et al. 2014; Enck et al. 2014; Graf et al. 2013; Hedin et al. 2014; Huang 59 et al. 2014, 2012; Milanova and Huang 2013] detect insecure flows from sources to secure sinks by 60 analyzing the flow of data in the program. Coarse-grained dynamic information flow approaches [Jia 61 et al. 2013; Nadkarni et al. 2016; Roy et al. 2009; Xiang and Chong 2021] reduce the writing effort of 62 annotations by tracking information at the granularity of lexically or dynamically scoped section 63 of code instead of program variables. By writing annotation, users can increase precision of the 64 information flow results [Xiang and Chong 2021]. Moreover, there are approaches using program 65 logic [Amtoft et al. 2006, 2008; Beckert et al. 2013] to analyze and reason about information flow. 66

In this work, we focus on security type systems for object-oriented languages [Banerjee and Nau-67 mann 2002; Barthe et al. 2007; Myers 1999; Strecker 2003]. Sun, Banerjee, and Naumann [Banerjee 68 and Naumann 2002; Sun et al. 2004] created a Java-like language annotated with security levels for 69 the standard information flow policy with only two security levels. Myers et al. [Myers 1999] created 70 the Jif language which extends Java with a type system that supports information flow control. 71 The precision of the type systems for object-oriented languages is a major challenge. Both related 72 approaches do not have an alias analysis or an immutability concept, so they conservatively reject 73 secure programs where confidential and non-confidential references could alias the same object. 74 This important drawback is addressed in our work. Additionally, as done for other type systems, 75 we give a correctness guarantee through a proof of noninterference: high data never influences 76 low data. This means that an attacker who can observe low data cannot obtain information about 77 high data. If an untrusted library is in the code base, the developer can leverage the type system to 78 ensure that only low data is served to such library. 79

We introduce SIFO which supports information flow control for an object-oriented language with 80 type modifiers for mutability and alias control [Giannini et al. 2019]. With respect to former work 81 on security type systems for object-oriented languages, SIFO provides a more precise type system, 82 allowing to type more correct programs. In this work, we show that reasoning about immutability 83 and encapsulation is beneficial to reason about information flow. In addition to adding expressivity, 84 SIFO allows a natural and compact programming style, where only a small part of the code needs 85 to actually be annotated with security levels. This result is achieved by building over the concept 86 of promotion/recovery [Giannini et al. 2019; Gordon et al. 2012], and extending it to allow methods 87 and data structures to be implicitly parametric on the security level. For example, with promotion, a 88 data structure can be used with any security level, but security is still enforced by not allowing data 89 structures of different security levels to interfere with each other. This reduces the programming 90 effort of developers and supports reuse of programs and libraries [Giannini et al. 2019]. 91

The contents of this paper are as follows. First, we introduce the language SIFO for information flow control. Second, we formalize the type system by introducing typing and reduction rules. Third, we show that our language is sound by proving the noninterference property that secret data is never observable by a public state. Fourth, we implement SIFO and evaluate it with a feasibility study and a benchmark to compare SIFO with state-of-the-art information flow analysis tools.

#### 2 INFORMAL PRESENTATION OF SIFO 99

100 In this section, we explain the challenges of securely checking the information flow in object-101 oriented languages. We then give an informal introduction to SIFO. Last, we discuss well and ill 102 typed SIFO expressions for a more detailed explanation. 103

#### Motivating example 2.1

104

105

106

108

117

118

119

120

121

129

130

131

1 107

Consider the following partially annotated code using two security levels low and high:

```
class Person { low id; Person(low id){ this.id=id; } }
   2
   3
      low local_id = GiveMe.anId();
<sup>109</sup> 4
      high p = new Person(local_id);
<sup>110</sup> 5
      high inside = p.id;
```

111 In SIFO, security is an instance based property: the person p is high, but other persons could have a 112 different security level. Security is also a deep property: the content of all the fields of p encodes 113 high information. In our example, every person has an id. Even if the id field is declared low, it will 114 encode high information inside of the high instance p. The value of local\_id is low. We can use it to 115 initialize id since information can flow from low to high, but not from high to low. 116

When extracting the value of the field id, the information is now part of the high Person, and thus, needs to be seen as high: p.id produces a high value.

Is this code conceptually correct with respect to information flow? Can we complete the type annotations on this code to make it correct? If the id is just a primitive integer, this is possible and easy in both SIFO<sup>2</sup> and other languages for information flow, such as Jif [Myers 1999]:

```
class Person { //SIF0 CODE
122 1
123<sup>2</sup>
        Int id; //low is the default security level
        Person(Int id){ this.id = id; }
   3
                                             }
124
   4
      . . .
125
   5
      local_id = GiveMe.anId(); //a low Int
126
      high p = new high Person(local_id);
   6
127
  7
      high inside = p.id;
128
```

The corresponding Jif code is also quite easy, but a little more involved and with a different syntax; we report it in Listing 1. Both in SIFO in Line 6, and in Jif in Line 9, the value of local\_id gets *promoted* from low to high.

```
132
      class Person[label L] {
133 1
         final int{L;this} id;
134 2
135 <sup>3</sup>
         Person(int{L; this} id){ this.id = id; }
136 4
      }
137 <sup>5</sup>
      . . .
      //label {high->low} allows low and high to read the variable
   6
138 7
      int{high->low} id = GiveMe.anId();
139 8
      //label {high->} allows only high to read the variable
140 g
      Person[{high->}]{high->} p = new Person[{high->}](id);
141<sub>10</sub>
       int{high->} inside = p.id;
142
                                         Listing 1. Example in Jif syntax
143
```

<sup>144</sup> <sup>2</sup>In the examples, we will use a rich language including local variables and literals with the usual semantics. Those are all 145 supported by our artifact, but in the formal model we present a minimal language where we keep only the most crucial OO features. 146

What happens if the id is a more complex custom object type? The following code is accepted in SIFO:

```
150
  1
      class Account { Int id; String name; Date firstTransaction;... }
<sup>151</sup> 2
      class Person {
152 3
        Account id;
        Person(Account id){ this.id = id; } }
153 4
154 5
      local_id = GiveMe.anId();//a low Account
155 6
      high p = new high Person(local_id);
156 7
      high inside = p.id;
  8
157
      high name = inside.name;
   9
158
```

As you can see, not much has changed. Of course, we need to define the Account class, but then we can use it in the same way we use Int before.

On the other hand, Jif [Myers 1999] (the most closely related work) cannot accept this kind of 161 code. In a pure object-oriented setting, everything is an object, and pre-defined types, as integers, 162 should be treated as any other object. However, Jif treats primitive types in a privileged way. In Jif, 163 it is possible to write more flexible code relying on primitive types than on objects. The difficulty 164 revolves around aliasing and mutation: the local variable local\_id is still available, and now it is 165 aliased inside of the high Person object p. Thus, if p is used to update any field of the Account, then 166 the low part of the program could see this high information through local\_id. This can happen 167 because with the base type system of Java all objects can be both mutated and aliased. Jif builds on 168 top of Java and only adds type properties directly related to information flow, so it cannot make 169 immutability and aliasing assumptions. On the other side, SIFO builds on top of L42 [Giannini et al. 170 2019], a language with built-in support for immutability and aliasing control using type modifiers 171 (also called reference capabilities). 172

In L42, the default modifier for references is imm (immutable), and the default security level of SIFO is low. Thus, a fully annotated version of the code above would look as follows:<sup>3</sup>

```
175
   1
      class Account {
176 2
         low imm Int id; low imm String name; low imm Date firstTransaction;
177 3
         ... }
178 4
      class Person {
         low imm Account id;
179 5
         Person(low imm Account id){ this.id = id; }
180 6
<sub>181</sub> 7
         }
182<sup>8</sup>
       . . .
183 <sup>9</sup>
      low imm Account local_id = GiveMe.anId();
10
184
      high mut Person p = new high Person(local_id);
  11
      high imm Account inside = p.id;
185
12
      high imm String name=inside.name;
186
```

Since the Account is immutable, in SIFO the value of local\_id is promoted from low to highfor the constructor call, exactly as it happens for Int before. Indeed, deeply immutable objects allow for the same kind of reasoning that primitive types allow in Java. In this way, SIFO code can scale and use objects as easily as primitive types in contrast to Jif and other approaches.

To make the same kind of behavior accepted in Jif, the code would have to be modified in the following way:

 <sup>&</sup>lt;sup>193</sup> <sup>3</sup>To help readability, in the rest of the paper we will write type modifiers and security levels explicitly, but a syntactically
 <sup>194</sup> much lighter style, as shown above, is accepted by our artifact and it is the preferred way to code, once the programmer
 <sup>195</sup> gets used to those defaults.

<sup>196</sup> 

```
197 1 low mut Account local_id = GiveMe.anId(); //in Jif, there is no immutability
198 2 low mut Date a = local_id.firstTransaction;
199 3 high mut Person p = new high Person(
200 4 new high Account(local_id.id,local_id.name,
201 5 new high Date(a.day,a.month,a.year)));
```

This code is accepted by both SIFO and Jif. This is a technique called *defensive cloning* [Bloch 2016]; it is very popular in settings where aliasing and mutability cannot be controlled.

In SIFO, we have *mutable* and *immutable* objects; where the reachable object graph (ROG) of an immutable object is composed only of other immutable objects (deep immutability), while the ROG of a mutable object can contain both mutable and immutable objects [Giannini et al. 2019]. The set of mutable objects in a ROG is called MROG.

In addition to imm, L42 also offers the capsule concept: a capsule reference refers to a mutable object whose MROG is reachable only through such reference. Both imm and capsule references can be safely promoted from low to high; this avoids the need of defensive cloning also when encapsulated mutable state is involved.

```
212
1 low capsule Account local_id = GiveMe.anId();
213
2 high mut Person p = new high Person(local_id);
214
```

Capsule variables are affine, that is, they can only be used zero or one time, thus if p is used to update the state of the account, the local capsule variable local\_id cannot be used to examine these updates.

As you can see from those examples, aliasing and mutability control is a fundamental tool 218 needed to support information flow in the context of an object-oriented language. A typical 219 misunderstanding of type modifiers is that a mutable field would always refer to a mutable object. 220 This is not the case, indeed all the fields of immutable objects will transitively contain only 221 immutable objects. This of course includes all fields originally declared as mutable. The same 222 applies to security labels: a low field in a high object would transitively contain only high objects. 223 This is different with respect to many other object-oriented languages, where the declarations 224 determine what to expect. If there is information from the context, that is normally explicit in the 225 usage site. In SIFO instead, the declared type is only a first approximation: the security level (and 226 type modifier) of an expression is a combination of what is declared in the class table and what is 227 implied from the usage site. 228

Note how in our example the class Person is declared with a low field, but the high Person object 229 actually stores a high value for such field. In SIFO, a low field is not like a low field in Jif, but it is 230 more like a field with generic/parametric security: the value of a low field of a low object will be low 231 but the value of a low field of a high object will be high. In general, the ROG of an object is always 232 at least as secure as the security of the object itself. This aligns nicely with mutability control in 233 L42, where immutable instances of classes declaring a mut field will hold immutable values in such 234 fields. Deep properties (like L42 immutability and SIFO security) allow for a much simpler and 235 more predictable reasoning with respect to (optionally) shallow properties, like Rust immutability 236 (that supports internal mutability) and Jif security (where low values can be stored in the ROG of 237 high values). 238

## 2.2 SIFO Concepts

Objects and references. As we anticipated above, in SIFO, we have *mutable* and (deeply) *immutable* objects. We also have four kinds of references: *imm, mut, capsule,* and *read* [Giannini et al. 2019]. An *imm* reference must point to an immutable object, and can be freely aliased, but as the name suggests, the fields of an immutable object cannot be updated. A *mut* reference must point to a mutable object;

1:5

245

239

Tobias Runge et al.

	Т	::=	s mdf C			
246	S	::=	high   low   (user defined)			
247	mdf	::=	mut imm capsule read			
248	CD		class C implements $\overline{C} \{\overline{F}   \overline{MD} \}$ interface C extends $\overline{C} \{\overline{MH}\}$			
249	ED F		C = C = C = C = C = C = C = C = C = C =			
250	I MD		$S = \{1, 1, 3, 1, 3, \dots, n\}$			
251	MD	=				
252	MH	::=	s maf method I $m(I_1 x_1, \ldots, I_n x_n)$			
253	е	::=	$x   e.f   e_0.f = e_1   e_0.m(e)  $ new $s C(e)$			
254						
255			Fig. 1. Syntax of the core calculus of SIFO			
256						
257	such an object can	be al	iased and mutated. A capsule reference points to a mutable object that is			
258	kept under strict a	liasin	g control: the mutable ROG reachable from the capsule reference cannot be			
259	reached from othe	r refe	rences. The capsule reference can be used only once to assign this isolated			
260	nortion of the hear	to a	reference of any kind. In particular, this means that a capsule reference can			
261	be used to initialize/undate an immaference/field, when this happens all the objects in the POC of					
262	such a reference become immutable					
263	The "enly yead	noo"	continuable.			
203	The only used (	1- : -	testriction is necessary so that no anas for the isolated portion of the neap			
204	can be introduced,	whic	n would violate the capsule property.			
205	Finally, a read re	feren	ce is the common supertype of imm and mut. With a read reference, the RUG			
266	cannot be mutated and aliases cannot be created; but there is no immutability guarantee that the					
267	object is not access	sible l	by other references, even mut ones.			
268						

Types. Types in SIFO are composed by a security level s, a type modifier *mdf* and a class name C. The security levels s are arranged in a lattice that specifies the allowed data flow direction between them. For example, we have a lattice with a low and a high security level, where the allowed information flow is from low to high, but not vice versa. The type modifier mdf can be mut, imm, capsule, and read as introduced above. The subtyping relation between modifiers is defined as follows: for all mdf, capsule  $\leq mdf$ ,  $mdf \leq$  read. This means that, for example, a mut reference can be used if a read is needed, and a capsule reference can be used both as a mut or as an imm one. This is sound, because capsule variables can be used only once.

277 Core Calculus. The syntax of the core calculus of SIFO is shown in Fig. 1. It covers classes C, field 278 names f, method names m, and declarations for classes, interfaces, and methods. A class consists 279 of fields and methods. The class itself has no modifier or security level. The modifiers and security 280 levels are associated with references and expressions. A field has a type T and a name. A method 281 has a return type, a list of input parameters with names and types, and also a security level and 282 a type modifier for the receiver; they are specified in front of the keyword method. We have the 283 standard expressions: variable, field access, field assignment, method call, and object construction. 284 When an object is initialized, its security level is initially determined by the constructor invocation. 285 Thus, different references to objects of the same class can have different security levels. C, m, f, 286 and *x* in Fig. 1 are all disjoint syntactic categories.

Method Calls. A method has to be defined in a class with parameter types, a return type and a 288 receiver type. For example, an Accumulator class has a low Int field acc and a method add that adds 289 another low Int parameter to the acc field and updates the field. 290

```
291 1
      class Accumulator {
292 2
        low imm Int acc;
        low mut method low imm Int add(low imm Int x){ return this.acc=this.acc+x; }
293 3
294
```

1:6

269

270

271

272

273

274

275

276

```
295 4
        }
       ...// below we show examples of user code
296 5
      low Accumulator a = GiveMe.aLowAcc();//a low Accumulator
297 6
<sub>298</sub> 7
      low imm Int w = 5;
299<sup>8</sup>
      low imm Int x = a.add(w); //ok to call as low*low -> low
   9
300
10
301
      high Accumulator b = GiveMe.aHighAcc();//a high Accumulator
  11
      high imm Int y = 6;
302
12
      high imm Int z = b.add(y); //ok to call as high*high -> high
303
```

We can call such a method if the receiver and the actual input parameter are low. However, SIFO 304 adds flexibility to method calls using *multiple method types*. Formally defined in Section 4, they 305 allow to call a method as if it was declared with a range of different type modifiers and security 306 levels. In this example, the multiple method types rule allows us to call the add method also if the 307 receiver and the parameter are all increased to the same security level (for example to high) and 308 returning a value with this same security level. Without this feature, the method add needs to be 309 declared for each security level. This feature also has benefits in comparison to a parameterized 310 version of the language because legacy code and standard libraries can be used in SIFO without 311 adding security level annotations: low is the default security level, and imm is the default modifier. 312

Control Flow and Implicit Information Leaks. Information flow control mechanisms [Sabelfeld and 314 Myers 2003; Volpano et al. 1996] are used to enforce an information flow policy that specifies the 315 allowed data flow in programs. A program can leak information *directly* through a field update. This 316 can be prevented by ensuring that no confidential data is assigned to a less confidential variable. 317 However, information can also flow *implicitly* through conditionals, loops, and (crucial in OO) 318 dynamic dispatch. For example, the chosen branch of a conditional reveals information about the 319 values in the guard. As shown from Smalltalk [Goldberg and Robson 1983], in a pure OO language, 320 dynamic dispatch can be used to emulate conditional statements and various forms of iterations 321 and control flow. Thus, our core language does not contain explicit conditional statements, but they 322 can be added as discussed in Section 4. Loops can be implemented through recursive method calls. 323

Therefore, SIFO only needs a secure method call rule to prevent implicit information flow leaks. 324 In a method call, information of the method receiver can flow to the return value and mutable 325 parameters. Thus, the security levels of the return value and mutable parameters have to be 326 equal or higher than the security level of the receiver. Consider for example the following code: 327 res=myValue.a0rB(a,b) If the method a0rB returns the first or the second parameter depending on the 328 dynamic type of myValue, we could use the result to identify information about myValue. Note how 329 this pattern is very similar to the Church encoding of Booleans. Similarly, if parameter a is typed 330 low mut and the receiver has a high security level, information of the high receiver can be leaked by 331 observing the mutations on the parameter a after the method call. 332

## 2.3 Examples of Well-Typed and Ill-Typed SIFO Expressions

In Listing 2, we show secure and insecure programming statements to explain the reasoning about information flow in SIFO.

A class Person contains a low imm String name and two high fields: a mut Password and an imm AccountId. The Password and AccountId class have a String field to set the actual password/id. When accessing a field, we consider the security level of both the field and the receiver and determine the least upper bound of both security levels in the lattice. When an object is initialized, it is created as mutable, and the initial security level is determined by the constructor invocation. For example, a Person can be initialized with a low or with a high security level.

343

333

334

```
class Person{low imm String name; high mut Password pwd; high imm AccountId id;}
_{344} 1
345<sup>2</sup>
      class Password{low imm String pwdS;}
346<sup>3</sup>
      class AccountId{low imm String idS;}
   4
347
   5
      low mut Person p =...//a pre existing low Person reference
348
      high mut Password pass = p.pwd;//ok, access of high Password
   6
<sup>349</sup> 7
      high imm String passS = p.pwd.pwdS;//ok, access is typed high
350 8
      low imm String passS = p.pwd.pwdS;//wrong, high assigned to low
351 q
      p.pwd.pwdS = highString;//ok, field update with high String
35210
      p.pwd.pwdS = p.name;//ok, as an imm String can be promoted
35311
      p.name = highString;//wrong, high String assigned to low p.name
35412
      high mut Person pHigh =.../a pre existing high Person
35513
356<sup>14</sup>
      pHigh.name = highString;//ok, field update with high String
357<sup>15</sup>
16
358
17
      low mut Password newPass = new low Password("some");//ok
      p.pwd = newPass;//wrong, mutable secret shared as low and high
359
18
      newPass.pwdS = "password";//ok? Insecure with previous line
360<sub>1</sub>9
<mark>361</mark>20
      low capsule Password capsPass=new low Password("secret");//ok
362<sub>21</sub>
      p.pwd = capsPass;//ok, no alias introduced
36322
      low imm AccountId aid = new low AccountId("secretId");//ok
36423
      p.id = aid;//ok, aid is imm and can be aliased
36524
      aid.idS = "0";//wrong, immutable object cannot be updated
36625
<mark>367</mark>26
27
368
      low read Person pRead = p;//ok, assigned to read reference
28
369
29
      high imm String passS = pRead.pwd.pwdS;//ok, access is high
      pRead.pwd.pwdS = highString;//wrong, read cannot be updated
370
30
      someMutObject.fieldName = pRead;//wrong; there are no read fields
371<sub>31</sub>
      someMutObject.fieldName = pRead.id;//ok if the field has an imm type
372
                                          Listing 2. SIFO examples
373
```

Consider the assignments in Listing 2 starting with Line 5 (line numbers are referenced in parentheses in the following): To ensure confidentiality, the type system prevents the password to be leaked via a low reference (8), but it can be exposed to another high reference (6, 7). The password can be updated with another high String (10), or with a low String (11), as we allow to promote the security level of imm references. The opposite of updating a low field of a low reference with a high String is forbidden (12), but the assignment is allowed if the reference and the String are high (14).

Until now, we explained assignments of immutable Strings; but the most interesting challenge 382 to guarantee confidentiality is about assigning mutable objects instead. For example, how can we 383 update the mutable p.pwd field? When a new Password object is created, it can be initialized as a low 384 object as the Password is not confidential on its own. The confidentiality is the association between 385 the Person object and the Password object. SIFO prevents that a low reference to a Password object is 386 assigned to a Person object (17). The reason is that the variable newPass is still in scope after the field 387 update, thus if (17) was accepted, (18) could be used to sneak a password change without the need 388 of any high information. 389

A secure assignment without aliases is shown in (20, 21). Here, the capsule modifier is utilized. A reference to a Password object can be assigned to a Person object, if the reference to the Password

1:8

374 375

390

object is high. The password is initialized by the programmer as low capsule. The system of type 393 modifiers is flexible enough to promote the created object from mut to capsule. Since there is no mut 394 395 value in the input, we are sure that an isolated portion of memory is created, as the created object cannot be accessed from any other mut reference. In (21), the flexible type system can then promote 396 the variable capsPass to a high capsule, which is assigned to the p.pwd field. 397

As discussed before, aliases over imm references (24), are allowed to move from lower security to 398 higher one. The alias does not lead to a security leak because the type system ensures that fields of 399 AccountId aid cannot be updated (25). Both imm and capsule references are referentially transparent, 400 and can be used as a controlled way to communicate between different security levels. 401

Finally, with read references (27), imm fields can still be accessed (28), but no fields can be updated 402 (29). Here, we present a simplified L42 type system, where fields can only be **mut** or **imm**; thus there 403 is no field that can be updated using a read reference (30). In the full L42, it is possible to have, for 404 405 example, read linked lists of read elements, but this has some subtle interaction with promotions, so we omit it here for simplicity. Of course, imm references reached from read references (31) can be 406 assigned to imm fields as usual. 407

For a more compelling example of our system that can promote expressions, consider the following listing:

```
class PassFactory{
411 1
412<sup>2</sup>
         low imm method low mut Password from(low imm String base){
   3
413
           low mut Password res = new low Password(base);
   4
414
   5
           if (this.tooSimple(base)){res.pwd = this.complete(base);}
   6
           return res;
<sup>416</sup> 7
         }
417<sub>8</sub>
      }
418 9
      p.pwd = passFactory.from("foo")
41910
```

The method from is well typed. The method from returns a low mut Password res which could not be directly assigned in (10) because a high security level is needed, but the system of type modifiers is flexible enough to promote low mut Password res to a high security level by utilizing the capsule modifier. Since there is no mut value in the input, we are sure that an isolated portion of memory is created (a capsule). With controlled aliasing, we can promote the capsule reference to a high security level (i.e., a low capsule Password can be promoted to a high capsule Password), and then, it can be assigned to the field in (10). All in all, a mut references can be promoted to a capsule, transferred to another security level and then reassigned to another mut reference.

Any method that takes a single mut in input, mutates it, and returns it as mut can be called with a capsule parameter and the result will also be promoted to capsule. This pattern allows great flexibility when encapsulated mutable objects need to be mutated [Giannini et al. 2019].

#### **DEFINITIONS FOR THE SIFO TYPE SYSTEM** 3 433

In this section, we define well-formedness of the type system and useful helper methods to introduce typing rules in the following section.

Well-Formedness. A well-formed program respects the following conditions: All classes and 437 interfaces are uniquely named. All methods in a specific class or interface are uniquely named. 438 All fields in a specific class are uniquely named. All parameters in a method header are uniquely 439 named, and there is no explicit method parameter called this. The subtyping graph induced by 440

441

408

409 410

415

420

421

422

423

424

425

426

427

428

429

430

431 432

434

435

1:10

446

447

448

449

450

451

452

453 454

455 456

457

458

459

460 461

462 463 464

465

466 467

468

469

470

471

472

473

474

475

476

477

- sec(T) = s, returns the security level s in type T.
- mdf(T) = mdf, returns the modifier in type *T*.
  - class(T) = C, returns the class C in the type T.
- fields(C) =  $T_1 f_1 \dots T_n f_n$ , returns the field declarations of class C.
  - p(C) = class C implements  $\overline{C} \{\overline{F} \ \overline{M}\}$ , returns the declaration of class C.
  - $lub(s_0, \ldots, s_n) = s$ , returns the least upper bound of the parameters  $s_0, \ldots, s_n$ .
  - T[s'] = lub(s, s') mdf C, where T = s mdf C, defined only if  $s' \le s$  or  $s \le s'$ ; returns a new type with security level lub(s, s')
  - mdf ▷ mdf' = mdf'', returns the modifier of an expression when accessing a field.
     mut ▷ mdf = capsule ▷ mdf = mdf
     imm ▷ mdf = mdf ▷ imm = imm
     read ▷ mut = read.

Fig. 2. Helper functions

If s mdf method T  $m(T_1 x_1 ... T_n x_n)$  is declared in C, with  $T_0 = s$  mdf C then 1:  $T_0[s'] ... T_n[s'] \to T[s'] \in methTypes(C, m)$ 2:  $(T_0[s'] ... T_n[s'] \to T[s'])[mut capsule] \in methTypes(C, m)$ 3:  $(T_0[s'] ... T_n[s'] \to T[s'])[read imm, mut capsule] \in methTypes(C, m)$ 

Fig. 3. Definition of multiple method types

implemented interfaces is acyclic (in this simplified language, we do not have class extension). capsule references can be used at most one time.

*Helper Functions.* In Figure 2, we show some helper functions for our type system. The first three notations extract the security level, the type modifier, and the class name from a type. The next two return fields and class declarations. The *lub* operator is defined to return the least upper bound of a set of input security levels arranged in a lattice. For example, since  $low \leq high$ , we have lub(low, high) = high. A lattice of security levels was first introduced by Bell and LaPadula [Bell and LaPadula 1976], and Denning [Denning 1976]. A lattice is a structure  $\langle L, \leq, lub, \top, \bot \rangle$  where *L* is a set of security levels and  $\leq$  is a partial order (e.g.,  $low \leq high$ ). The lattice defines an upper bound of security levels. A set of elements  $X \subseteq L$  has an upper bound *y* if  $\forall x \in X : x \leq y$ . An upper bound *u* of *X* is the least upper bound (*lub*) if  $u \leq y$  for each upper bound *y* of *X*. To form an upper semi-lattice, a unique least upper bound (*lub*) for every subset of *L* must exist. Additionally, we restrict the lattice to be bounded with the greatest element  $\top$  and the least element  $\bot$ .

In Figure 2, the function  $s \ mdf \ C[s']$  returns a new type whose security level is the least upper bound of the two. The security level is set to lub(s, s') and the modifier and class remain the same. The last function  $mdf \triangleright mdf'$  computes a resulting modifier if a field with type modifier mdf' is accessed from some reference with type modifier mdf. For example, if we access a mut field from a mut reference we get a mut value, but if we access a mut field from a read reference, we get a read value. If either the reference or the field are imm, then imm is returned; thanks to deep immutability, the whole reachable object graph is immutable.

*Multiple Method Types.* Instead of a single method type as in Featherweight Java [Igarashi et al. 2001], we return a set of method types using  $methTypes(C, m) = \{\overline{T_0} \rightarrow T_0, \ldots, \overline{T_n} \rightarrow T_n\}$ . The programmer just declares a method with a single type, and the others are deduced by applying all the transformations shown in Fig. 3. Multiple method types reduce the need of implementing

the same functionality several times, where the same parameter has only different type modifiers 491 or security levels. The base case, as declared by the programmer, can be transformed in various 492 ways: (1) A method working on lower security data can be transparently lifted to work on higher 493 security data (some security level s'). This means that methods that are not concerned with security 494 are usually declared as working on a low receiver and low parameters, returning a low result. Note 495 that the security level remains unchanged when s' is chosen as low. Thus, we can use T[s'] with s' 496 different from **low** when we are in a context where we need to manipulate secure data. The multiple 497 method types lift the method as if it was declared with higher receiver, parameters, and return type. 498 For example, a mathematical method should return the same security level as the security level of 499 the parameters. In our language, we can just implement this method once with the lowest security 500 level and reuse it with any other security level of the lattice. As a comparison, the Jif tutorial<sup>4</sup> 501 suggests that a mathematical method should be implemented with a generic security level. 502

(2) The second case swaps all mut types for capsule ones. If we provide capsule instead of mut in
the input, we can use the method to produce a capsule return value. This corresponds to capsule
recovery/promotion in [Giannini et al. 2019]. By providing all mut parameters as capsule, the method
would not take any mut as input. Any mut object that is returned, is created inside of the method
execution (as we do not have any form of global state/variables) and thus can be seen as a capsule
from outside the scope of the method body. For example, a method declared as

509 low mut method low mut Person father(low imm String name)

510 can be also used as if it was declared as

511 low capsule method low capsule Person father(low imm String name),

where all **mut** parameters (just the receiver in this example) and the return type are turned into 512 capsule. (3) The third case swaps all mut types for capsule ones and all read types for imm ones. This is 513 useful if the method was returning a read value; in this case we can obtain an imm. This corresponds 514 to immutable recovery/promotion in [Giannini et al. 2019] and can be intuitively understood by 515 considering that a read reference can point to either an immutable or a mutable object. If it was an 516 immutable object, it is fine to return it as imm; if it was a mutable object, then for the same reasons 517 as case (2), we can promote it to capsule, which is a subtype of imm. Note that in all the three cases, a 518 519 method working on lower security data can be transparently lifted to work on higher security data.

### 4 TYPING RULES

The typing rules are presented in Fig. 4. We assume a reduction similar to Featherweight Java [Igarashi et al. 2001; Pierce 2002]. We have a typing context  $\Gamma ::= x_1 : T_1 \dots x_n : T_n$  which assigns types  $T_i$  to variables  $x_i$ .

- SUB and SUBSUMPTION. We allow traditional subsumption for modifiers and class names.
   However, we are invariant on the security level. We assume our interfaces to induce the standard subtyping between class names.
- <sup>528</sup> **T-VAR.** A variable *x* is typed using the context  $\Gamma$ .
- FIELD ACCESS. The result of the field access has the class of the field f. The security level is the least upper bound of the security levels of  $e_0$  and f. The resulting modifier is the sum of the modifiers of  $e_0$  and f as defined in Fig. 2. In this way, if we read a low mut Person field from a high read receiver, we obtain a high read Person result.
- FIELD ASSIGN. The reference resulting from  $e_0$  has to be mut to allow the assignment. The security level of the assigned expression  $e_1$  is the least upper bound of the security levels of expression  $e_0$  and the field f as declared in C. For example, if we assign a high expression  $e_1$ , either the field f or the reference resulting from  $e_0$  need to be high.

<sup>538</sup> <sup>4</sup>https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html

ACM Trans. Program. Lang. Syst., Vol. 1, No. 1, Article 1. Publication date: January 2022.

539

520

Tobias Runge et al.

#### Fig. 4. Expression typing rules

**CALL.** We allow a method call if there is a method type where all parameters and the return value are typable. The security levels of the return type and all **mut** or **capsule** parameters have to be greater than or equal to the security level of the receiver. This requirement is needed because through dynamic dispatch the receiver may leak information. This is one of the crucial points of our formalism, as explained in Section 2.2.

**NEW.** The newly allocated object is created as a mutable object, and with a specified security level s. This rule checks that the parameter list  $e_1 ldots e_n$  has the same length as the declared fields. The object of class C has a list of fields  $f_1 ldots f_n$ . Each parameter  $e_i$  is assigned to a field  $f_i$ . This assignment is allowed if the type of parameter  $e_i$  is (a subtype of)  $T_i[s]$ . The programmer can choose s to raise the expected security level over the level originally declared for the fields. In order to use an actual parameter with a higher security level (s) to initialize a field defined with a lower security level, the newly created object needs to have this chosen security level s. By using rule SEC-PROM, we can do the opposite, initializing higher security fields with lower security imm/capsule values. For example if we have a class Ex \{low Object a; high Object b; topSecret Object c; \}, we can create correct objects with

- 582 1 new low Ex(lowValue, highValue, topSecretValue)
- 583 2 new high Ex(highValue, highValue, topSecretValue)

584 3 new topSecret Ex(topSecretValue, topSecretValue, topSecretValue)

An object new high Ex(highValue, topSecretValue, topSecretValue) is incorrect because the sec ond parameter is topSecret. Accessing a high reference and a high field would return a high
 value. This leaks the topSecretValue object.

ACM Trans. Program. Lang. Syst., Vol. 1, No. 1, Article 1. Publication date: January 2022.

1:12

589  $\Gamma \vdash e : s \text{ imm Bool}$ 

590

591

593

594 595

596 597 598

599

600

617

618

619

620

621

622

623

624

625

626

627

628

629

630

637

 $\frac{\Gamma[mut(s), final(s)] \vdash e_1 : T \qquad \Gamma[mut(s), final(s)] \vdash e_2 : T}{\Gamma \vdash if \ e \ then \ e_1 \ else \ e_2 : T}$   $\frac{\Gamma \vdash e : s \ mdf \ C \qquad mdf \in \{inm, capsule\}}{\Gamma \vdash declassify(e) : \perp mdf \ C} \qquad (Decl)$ 

Fig. 5. Extension: expression typing rules for if and declassify

The object new topSecret Ex(highValue, topSecretValue, topSecretValue) is only correct if the highValue object can be promoted to topSecret. This is only possible for immutable or encapsulated objects.

- PROM. Promotion from mut to capsule is possible if all mut references are only seen as read in
   the typing context. Since a read cannot be saved into a field of a mut object, we know that
   the reachable object graph from those read variables will not be part of the reachable object
   graph of the result.
- 605 SEC-PROM. Security promotion raises the security level of a capsule or imm expression. This 606 captures the intuitive idea that a higher security level is allowed to see all data with lower 607 security levels. However, this is sound only for capsule or imm expression. An immutable 608 object cannot be modified, so the promotion is secure because no new confidential informa-609 tion can be injected into its ROG. Also a capsule object can be passed to a new reference with 610 a higher security level. A leak through the lower capsule reference cannot happen, because 611 a capsule reference can be used at most one time. Instead, in the case of a mutable object, 612 this assignment would cause a possible leak: it would allow a high and low alias reference 613 to the same object; and if the high reference was updated with high data, the low reference 614 would see such data as well. Also read cannot be promoted: a promoted read object can have 615 a low mut alias reference to the same object that could now sneakely update the data seen as 616 high.
  - **M-Oκ.** This rule checks that the definition of a method is well typed. Using the receiver type and the parameter types, *e* must have the same type as the declared return type.
  - **C-OK.** This rule checks that the definition of a class is well typed. The rule uses a helper function *mhs* which returns the method headers declared in a set of classes or interfaces, or it directly returns the headers of a set of methods. A well typed class *C* implements all methods that are declared in the interfaces  $\overline{C}$ .
    - I-OK. A correct interface must contain all method headers of the implemented interfaces *C*.

*Implicit Information Flows.* The language as presented is minimal but using well-known encodings it can support imperative update of local variables (use box objects with a single field and field updates) and conditionals (use any variation of the Smalltalk [Goldberg and Robson 1983] way to support control structures). However, in Fig. 5, for the sake of a more compelling explanation, we show how the *if* construct could be typed if we expand our language with *if*, Booleans and updatable local variables.

**IF.** For a conditional statement with a Bool expression in the guard, we define that both branches  $e_1$  and  $e_2$  must have the same type T. With  $\Gamma[mut(s), final(s)]$ , we introduce two restrictions: with mut(s), we prevent mutation of mut objects with a security level lower than s by seeing them as read, and with final(s), we prevent updating all local variables with a security level lower than s. If s is the lowest security level, both functions do not restrict  $\Gamma$ .

 $(I_F)$ 

```
interface I {
638 1
639 <sup>2</sup>
         low mut method low imm I doIt(low imm I l1, low imm I l2);
640<sup>3</sup>
      }
   4
      class C1 implements I {
641
   5
         low mut method low imm I doIt(low imm I l1, low imm I l2) {
642
   6
           return l1; }
<sup>643</sup> 7
      }
644<sub>8</sub>
      class C2 implements I {
645 9
         low mut method low imm I doIt(low imm I l1, low imm I l2) {
           return 12; }
64610
      }
64711
64812
       . . .
      high mut C1 c1 = new high C1();
649<sup>13</sup>
650<sup>14</sup>
      high mut C2 c2 = new high C2();
651<sup>15</sup>
      low imm C1 l1 = new low C1();
16
652
17
      low imm C2 12 = new low C2();
      high mut I i = c1;//c1 assigned to reference of type I
653
18
      low imm I x = i.doIt(l1,l2);//ILL TYPED in our system
<mark>654</mark>19
      //if it was accepted, by observing low variable x = 11
655<sub>20</sub>
      //we could deduce the content of the high variable i = c1
65621
      //equivalent behaviour using an if
6572.2
      if (i instanceof C1){ x = l1; } else { x = l2; }
65823
659
                                    Listing 3. Ill-typed example of a method call
```

Therefore, assignments to less confidential variables or fields in the branches are prohibited to prevent leaks. This means that if the expression in the guard of a conditional statement has a security level that is higher than the lowest security level, only assignments to variables of at least the security level of the guard are allowed. Additionally, only mutable objects of at least the security level of the guard can be mutated. In this way, only data whose security level is at least the one of the guard can be mutated.

Using the Smalltalk-style as discussed above, our pre-existing rules would handle the encoded code exactly as with the explicit IF-rule, Booleans, and local variables. Thus, our system is minimal, but does not lack expressiveness. The IF-rule has a similar constraint as the CALL-rule where return type and mut and capsule parameters have to be greater than or equal to the security level of the receiver.

The following example, in both OO style and with an explicit if shows the mechanism to prevent 673 implicit information flow leaks: In Listing 3, we have an interface I that declares a method doIt 674 that gets two low imm I parameters as input. The classes C1 and C2 implement the interface and the 675 method. The implemented method of C1 returns the first parameter and the implemented method 676 of C2 returns the second parameter. We initialize two high variables C1 c1 and C2 c2 and two low 677 variables C1 11 and C2 12. By assigning c1 to i in Line 17, we hide the information of the explicit class 678 behind the interface. However, if we are allowed to execute Line 18, calling the method doIt, the 679 class c1 is revealed because 11 is returned. The attacker gets the information about the explicit class 680 by observing the return value. This example is an object-oriented implementation of a conditional 681 statement. If we use an if instead of dynamic dispatch, the leak is clearly visible (see Line 23), 682 as we assign low variables in branches of a high guard. To prevent such leaks, we define that the 683 security level of the return type is never lower than the security level of the receiver. Information 684 does not only flow through the method result: also **mut** or **capsule** parameters can be used to push 685

1:14

information out of the method; thus also the security level of mut and capsule parameters must never
be lower than the security level of the receiver. Thus, in both cases with the CALL-rule and with
the IF-rule the implicit leak is prevented.

Declassification. A mut or read reference can only be assigned to references of the same security 691 level. However, since imm and capsule references are referentially transparent, it is safe to assign 692 an imm or capsule object from a lower security level to a higher one. On the other hand, we are 693 not allowed to assign an object with a higher security level to a reference with a lower security 694 level. Similar to the if, we consider adding a declassify operator that can be used to manipulate the 695 security level s of expressions to allow a reverse information flow in appropriate cases. In some 696 cases, this reverse information flow is needed to develop meaningful programs. For example, if 697 a confidential password is hashed, the value should be assignable to a public output. With the 698 declassify expression, the security level of a capsule or imm reference is set to the lowest security 699 level. In Fig. 5, the DECL rule is shown. 700

**DECL.** The declassify rule is used to change the security level of capsule or imm expressions to the bottom level of the lattice.

declassify should be used with caution because secure information is leaked in the case of 704 inappropriate use. In this rule, we cannot declassify an expression to a specific security level, but 705 this is not a limitation, since we can encapsulate declassification statements inside of methods which 706 directly promote the declassified expression to the desired security level. mut and read references 707 cannot be declassified because potentially existing aliases are a security hazard. For example, if you 708 declassify a high mut reference and a high read alias still exists, an attacker could use that now low mut 709 reference to mutate information visible as high. Declassification is not part of our system to type 710 check secure programs, and we do not need it to make secure programs, rather it is a mechanism to 711 break security in a controlled way. That is, when comparing with examples of other papers [Myers 712 1999], we do not use declassify to encode behavior. We can still use it to print out results to show 713 that the code is working. In the DECL rule that we present, declassify is just a special expression. In 714 the full SIFO language embedded in L42, declassification can be flexibly tuned to the user needs 715 preventing accidental declassification. 716

## 5 PROOF OF NONINTERFERENCE

In this section, we aim to ensure *noninterference* [Goguen and Meseguer 1982] according to our 719 information flow policy. Noninterference is a central criterion for secure information flow, as we 720 want to ensure that an attacker cannot deduce confidential data by observing data with lower 721 security levels. It is based on the indistinguishability of program states. Two program states are 722 indistinguishable (also referred to as observably similar) up to a certain security level if they agree 723 on their memory reachable from references with a security level lower than that specific security 724 level. Using this property, a program satisfies the noninterference Theorem 5.0 if and only if the 725 following holds: if a program is executed in two observably similar memories up to a certain 726 security level, then the resulting memories are also observably similar up to the same security level, 727 but may differ in higher security levels. 728

729 730

717

718

690

701

702

703

THEOREM 5.0 (GENERAL NONINTERFERENCE).

If we have expressions  $e_1$  and  $e_2$  without declassification that are well typed and have the same low values, but possible different high values ( $e_1$  lowEqual  $e_2$  see Definition 5.6),  $M_1$  and  $M_2$  are well typed memories,  $M_1$  and  $M_2$  are low observably similar,  $M_1|e_1 \rightarrow^* M'_1|v_1, M_2|e_2 \rightarrow^* M'_2|v_2$ , then  $M'_1$  and  $M'_2$ are low observably similar, and memories  $M'_1, M'_2$ , values  $v_1$ , and  $v_2$  are well typed and  $v_1$  lowEqual  $v_2$ .

Tobias Runge et al.

754

755

756

757 758

770

771

772

773

784

 $e \quad ::= \quad x \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid \text{new } s \ C(\overline{e}) \mid v$   $\mathcal{E}_{v} \quad ::= \quad [] \mid \mathcal{E}_{v}.f \mid \mathcal{E}_{v}.f = e \mid v.f = \mathcal{E}_{v} \mid \mathcal{E}_{v}.m(\overline{e}) \mid v.m(\overline{v} \ \mathcal{E}_{v} \ \overline{e}) \mid \text{new } s \ C(\overline{v} \ \mathcal{E}_{v} \ \overline{e})$   $\mathcal{E} \quad ::= \quad [] \mid \mathcal{E}.f \mid \mathcal{E}.f = e \mid e.f = \mathcal{E} \mid \mathcal{E}.m(\overline{e}) \mid e.m(\overline{e} \ \mathcal{E} \ \overline{e'}) \mid \text{new } s \ C(\overline{e} \ \mathcal{E} \ \overline{e'})$   $v \quad ::= \quad s \ mdf \ o$   $M \quad ::= \quad o_{1} \mapsto C_{1}(\overline{o_{1}}) \dots o_{n} \mapsto C_{n}(\overline{o_{n}})$ 



In this section, we prove noninterference for a lattice with a low and a high security level (low  $\leq$  high) for terminating programs. Nonterminating programs and programs with an arbitrary lattice do typecheck, and we expect noninterference to work for those too, but our proof technique does not address those cases. We also do not include the declassify operation because this rule explicitly allows high data to interfere with data of lower security levels. This means, declassify is only an explicit mechanism to break security in a controlled way, as we explained in Section 2. In the section, we use the notation of memory and expression. The meaning of these terms are overloaded. A memory is often a stack in a while languages, in object-oriented languages, the memory is often a heap. In our expression-based language, we model a heap (a map from memory locations to the values stored for each field as defined by the location's class), and we use the expressions themselves to track the values that are accessible within each expression (similar to how the stack achieves this during the execution). This means, memory *M* captures the heap and expression.

#### 5.1 Reduction Rules

759 SIFO is an additional type system layer and does not influence the language semantics. However, 760 for the sake of the noninterference proof, we need to instrument the small-step reduction to keep 761 track of security and modifiers during program execution. To this aim, we define in Fig. 6 values 762 v as a location o in a store with security level and type modifier. The store is some memory M. 763 In M, a location points to some class C where each field is again a location  $o_i$  in the memory M. 764 With the evaluation context  $\mathcal{E}_{v}$ , we define the order of evaluation. We assume two well-formedness 765 properties. The memory is well-formed if M is a map from o to  $C(\overline{o})$ , thus the locations  $\overline{o}$  in domain 766 of M are unique. The reduction arrow  $(M|e \to M'|e')$  is well-formed if M and M' have no dangling 767 pointers with respect to e and e' (i.e., every pointer points to a valid object in the memory). In 768 Figure 7, the following reduction rules are shown. 769

**CTX.** This is the conventional contextual rule, allowing the execution of subexpressions.

- **FIELD ACCESS.** A field access  $f_i$  of a value v is reduced to a location  $o_i$  if the location o of v points to the suitable class  $C(o_1 \dots o_n)$ . The security level is the least upper bound of the security levels of v and  $f_i$  and the modifier is the sum of modifiers of v and  $f_i$ .
- **FIELD UPDATE.** The store  $o \mapsto C(\overline{o} \ o_0 \dots o_n)$  is updated with an assignment of  $v_2$  to the field  $f_0$ . The location  $o_0$  is replaced with o'. The security level of the resulting value  $v'_2$  is the least upper bound of the security levels of expression  $v_0$  and the field  $f_0$  as declared in *C*. The type modifier of  $v'_2$  is equal to the type modifier of the field  $f_0$  as declared in *C*.
- **CALL.** We reduce a method call to an expression *e*, where each value  $v_i$  is assigned to a parameter  $x_i$ . As we use multiple method types, the actual assigned values  $v'_i$  can have an updated security level or type modifier. Additionally, the called method has to be declared in the class  $C_0$  pointed to by the location in  $v_0$ . The concrete calculation of the updated types  $T'_i$  with mostSpecMethType is shown in Definition 5.1. The definition calculates the exact method types to support the proof of noninterference. To calculate the types  $T'_i$ , the

833

1:17

Tobias Runge et al.

834 
$$M|(\operatorname{high} mdf o).m(\overline{v}) \to^{+} M'|(s \ mdf \ o')$$

1:18

837 838 839

847

848

849

850

851

(HREC)  $\overline{M|\mathcal{E}_{v}[(\operatorname{high} mdf o).m(\overline{v})]} \rightarrow_{call} M'|\mathcal{E}_{v}[(\operatorname{high} mdf o')]$  $e \text{ not of form } \mathcal{E}_v[(\text{high } mdf \ o).m(\overline{v})] \qquad M|e \to M'|e'$ (HOTHER)  $M|e \rightarrow_{call} M'|e'$ 

Fig. 8. Additional reduction rules for the noninterference proof

parameters. It is allowed that formal parameter have higher security levels than s if a current parameter is passed with the same security level higher than s. With raiseFormalSecurity, it is checked that the actual parameters have the same security level as the now raised formal parameters. Only actual parameters with type modifier imm or capsule can be raised to the needed level. In the last step, the combined type modifier of the actual and formal parameter is calculated with the function of Fig. 2.

- 852 **NEW.** The newly created object is reduced to a location *o* in the memory that points to the 853 class C where each field is again a location  $o_i$ . The reduced value has the same security level 854 s as in the expression before the reduction and a mut type modifier. Note that we do not 855 need to say that a new reference is not in the domain of the old memory. This is implicit by 856 the well-formedness of the memory.
- 857 **HREC**, HOTHER. The two rules in Fig. 8 are added to condense the reduction of methods 858 calls on high receivers ( $\rightarrow_{call}$ ). These rules only consider a low  $\leq$  high lattice of security 859 levels and are needed for our noninterference proof for technical reasons. We will prove 860 noninterference only for a low  $\leq$  high lattice, but the typing rules work with any security 861 lattice. As you can see, we condense all execution steps that happen under the control of a 862 high receiver into a single more abstract step. Here,  $\rightarrow^+$  is the transitive closure. Of course, 863 rule (HREC) is not applicable when the reduction of such a method does not terminate; this 864 is why our proof technique only works on terminating programs: it does not make sense to 865 talk about noninterference for a program stuck into a non-terminating loop inside of a high 866 method. This program would never reach again a state where a low attacker may attempt to 867 observe data. 868

#### Definition of Similarity w.r.t. Security Levels 5.2

871 In this subsection, we define the (observable) similarity of two memories for the proof of non-872 interference. Therefore, we need further definitions of reachable object graphs (ROG). We define 873 mutable, low, and high memories using a notation of reachable object graph. Definition 5.2 defines 874 the mutable ROG of a memory and the current state of the main expression e. Note how this is similar 875 to ownership work where the stack is used to track the current top level state. In an expression-876 based language, the stack is represented as the set of values inside of the main expression. The MROG contains every location inside of the expression *e* that has a mut or a capsule type modifier. 878 Additionally, mut fields of locations in the MROG are included. In Definition 5.3, the low ROG contains 879 every location inside of the expression e that has a low security level. Additionally, low fields of 880 locations in the low ROG are included.

881 882

877

003	Definition 5.2. $MRog(M, e) = \overline{o}$	Definition 5.3. $lowRog(M, e) = \overline{o}$
885	• $o \in MRog(M, e)$ if:	• $o \in lowRog(M, e)$ if:
886	$e = \mathcal{E}[s \text{ mut } o]$	$e = \mathcal{E}[low\_o]$
887	• $o \in MRog(M, e)$ if:	• $o_i \in lowRog(M, e)$ if:
888	$e = \mathcal{E}[s \text{ capsule } o]$	$o \in lowRog(M, e)$
880	• $o_i \in MRog(M, e)$ if:	$o \mapsto C(o_1 \dots o_n)$ in M
890	$o \in MRog(M, e)$	class $C \subseteq \{T_1f_1 \dots T_nf_n\}$
891	$o \mapsto C(o_1 \dots o_n)$ in M	$sec(T_i) = 1ow$
892	$\{\texttt{class } C \ \ \{T_1f_1 \dots T_nf_n \}$	
893	$mdf(T_i) = mut$	

In Definition 5.4, the high ROG includes locations of high mut values. The high ROG also includes 894 895 every mut value that is pointed to by a location in the high ROG. Furthermore, the high mut fields of a location in the low ROG are included. We exclude high imm values, as imm values can be referenced by 896 both low and high references. 897

898	Definition 5.4. $highRog(M, e) = \overline{o}$	
899	• $o \in highRog(M, e)$ if :	• $o_i \in highRog(M, e)$ if :
900	$e = \mathcal{E}[high mut  o]$	$o' \in lowRog(M, e)$
901	• $o_i \in highRog(M, e)$ if :	$o' \mapsto C(o_1 \dots o_n)$ in M
902	$o' \in highRog(M, e)$	class $C \subseteq \{T_1f_1 \ldots T_nf_n\}$
903	$o' \mapsto C(o_1 \dots o_n)$ in M	$sec(T_i) = high$
904	class $C = \{T_1 f_1 \dots T_n f_n\}$	$mdf(T_i) = mut$
905	$mdf(T_i) = mut$	$o_i \in dom(M)$

907

908

909

910

911

912

913

914

915

916

In Definition 5.5, we define the observable similarity of two memories. Two memories  $M_1$  and  $M_2$  are similar given an expression e, if and only if the **low** locations of both memories are equal. As explained before, the expression *e* is needed to track the top level state. We need a transformation on the memories to filter high locations and high fields of classes pointed by low locations. As we are only interested in similarity of low locations for the noninterference proof, these high locations have to be filtered. The filtering is defined by  $M[only \overline{o}]$ . In the given memory M, each location o is removed that is not in the input set  $\overline{o}$ . Additionally, for the o in the input set  $\overline{o}$ , each  $o_i$  in  $o \mapsto C(o_1 \dots o_n)$ , where the corresponding field in class C is defined with a high security level, is replaced with the location o to filter the explicit high location  $o_i$ . Thus, with this transformation two memories are equal, if they differ only in the high ROG.

Removing the high locations in this way may produce an ill type memory; this is not a problem since those memories are only used as a device to define similar(e) and not in the reduction.

#### 917 918 Definition 5.5. 919 $M_1 similar(e) \ M_2 \leftrightarrow M_1[only \ lowRog(M_1, e)] = M_2[only \ lowRog(M_2, e)]$ 920 where $M[only \overline{o}] = M'$ is defined as: 921 • $(o_1 \mapsto C_1(\overline{o_1}) \dots o_n \mapsto C_n(\overline{o_n}))[only \overline{o}] =$ 922 $o_1 \mapsto C_1(\overline{o_1})[only \ \overline{o}] \dots o_n \mapsto C_n(\overline{o_n})[only \ \overline{o}]$ 923 • $(o \mapsto C(\_))[only \overline{o}] = empty if o is not in \overline{o}$ 924 • $(o \mapsto C(o_1 \dots o_n))[only \overline{o}] = o \mapsto C(o'_1 \dots o'_n)$ if $o \text{ in } \overline{o}$ with: 925 $fields(C) = T_1 f 1 \dots T_n f n$ 926 $o'_i = o_i$ if $sec(T_i) = 1ow$ 927 $o'_i = o \ if \ sec(T_i) = high$ 928 The Definition 5.6 compares that two expressions are equal if we only consider the low values. It 929 is defined as rule induction, where the only interesting case is that high values are ignored. This 930 931 ACM Trans. Program. Lang. Syst., Vol. 1, No. 1, Article 1. Publication date: January 2022.

means, two expressions are *lowEqual* if either they are the same *low* expression, or possibly-different
 high locations.

Definition 5.6. e lowEqual e'

- - e.f lowEqual e'.f iff e lowEqual e'
- $e_0.f = e'_0$  lowEqual  $e_1.f = e'_1$  iff  $e_0$  lowEqual  $e_1$  and  $e'_0$  lowEqual  $e'_1$ 
  - $e_0.m(e_1...e_n)$  lowEqual  $e'_0.m(e'_1...e'_n)$  iff  $e_i$  lowEqual  $e'_i$  for i in 0...n
  - new s  $C(e_1 \dots e_n)$  lowEqual new s  $C(e'_1 \dots e'_n)$  iff  $e_i$  lowEqual  $e'_i$  for i in  $1 \dots n$
  - (low mdf o) lowEqual (low mdf o)
  - (high mdf o) lowEqual (high mdf' o')

The Definition 5.7 is essential to constrain reduction: an alternative to our reduction that is 943 944 undesirable could trivially preserve security by adding everything to the high memory so that no low objects remain. The definition of *preserves* constraints the reduction to only change the high 945 memory in the few appropriate and necessary cases as follows: In the first trivial case, nothing is 946 added. In the second case, a new high object is created and added to the high ROG. In the third case, 947 a low capsule object is promoted to high and added to the high ROG. Here, all options of a promotion 948 of a capsule object are shown (field assign, method call as receiver and as parameter, and object 949 construction as a parameter). 950

Definition 5.7. *M* preserves(e/e') *M'* if one of the three holds:

- <sup>952</sup> 1) highRog(M, e) = highRog(M', e') <sup>953</sup> 2) highRog(M, e) = highRog(M', e')
- 2) highRog(M, e), o = highRog(M', e') then

 $e = ctx_0 [new high C(\overline{v})], e' = ctx_0 [high mdf o]$ 3) high Pog(M a) MPOC(M high mdf'a) = high Pog(M'a') then

3) highRog(M, e), MROG(M, high mdf' o) = highRog(M', e') then

 $e = ctx_0[low \ capsule \ o], \ e' = ctx_1[high \ mdf' \ o]$ 

and *e* is equal to one of the following:

 $ctx[v.f = low \ capsule \ o]$ 

 $ctx[(low capsule o).m(\overline{v})],$ 

 $ctx[v.m(\overline{v}(low\ capsule\ o)\overline{v'})],$ 

 $ctx[new C(\overline{v}(low \ capsule \ o)\overline{v'})]$ 

## 5.3 Noninterference Theorem and Proof

We prove noninterference according to our information flow policy. In literature, there are many proposed languages (with proofs) that are very similar to the type system proposed in this work [Giannini et al. 2019]. Here, to avoid repeating those same proofs that are already presented in those other works, we accept two assumptions. (1) Soundness: the reduction does not get stuck. (2) Immutability and encapsulation: In addition to not getting stuck, the reduction also never mutates the ROG of an immutable object, and the ROG of capsules is always encapsulated (i.e., all mutable objects can be reached only through the capsule reference). Both assumptions are established and proved before [Giannini et al. 2019].

To prove noninterference, we first introduce two lemmas that facilitate the proof. We show that the reduction terminates using  $\rightarrow_{call}$  and we show that, given two similar memories, each reduction step results in similar memories.

*Call-Reduction Termination.* We prove in Lemma 5.8 that the reduction  $\rightarrow_{call}$  does not interfere with termination: if we have a well typed memory *M* and an expression *e* and the program terminates with the normal reduction, then it also terminates with  $\rightarrow_{call}$ . The result for both reductions is also the same.

934

939

940

941

942

951

955

956

957

958

959

960

961

962 963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

LEMMA 5.8 ( $\rightarrow_{call}$  TERMINATION). If memory M and expression e are well typed and if the reduction terminates with  $\rightarrow$ , then it terminates also with the reduction  $\rightarrow_{call}$  with the same result.

PROOF. This can be verified by cases:  $\rightarrow_{call}$  behaves exactly like  $\rightarrow$ , but has a different granularity of the steps. Therefore,  $\rightarrow_{call}$  terminates in every case where  $\rightarrow$  terminates.

Bisimulation. To establish noninterference, Lemma 5.9, the bisimulation core, states that two 986 well typed and similar memories  $M_1$  and  $M_2$  and expressions  $e_1$  and  $e_2$ , where  $e_1$  lowEqual  $e_2$  holds, 987 reduce to  $M'_1|e'_1$  and  $M'_2|e'_2$  that are also similar, and the reduced expressions  $e'_1$  and  $e'_2$  are lowEqual. 988 We need property (2) to state that both memories are observably similar, and property (3) that also 989 the expressions are similar regarding the observable values. Both properties together represent 990 observably similar memories as in Theorem 5.0. Furthermore, the preservation property of each 991 memory ensures that the reduction only changes the high memory in necessary cases. With this 992 993 lemma, we know that each reduction step from similar memories results in similar memories: each  $\rightarrow_{call}$  reduction step ensures noninterference. 994

LEMMA 5.9 (BISIMULATION CORE).

Given well typed memories and expressions without declassification  $M_1$ ,  $M_2$ ,  $e_1$ , and  $e_2$ where  $M_1|e_1 \rightarrow^* |v_1|$  and  $M_2|e_2 \rightarrow^* |v_2|$ . If the following holds 1)  $M_1|e_1 \rightarrow_{call} M'_1|e'_1$ ,

2)  $M_1$  similar( $e_1$ )  $M_2$ ,

1001 3) and  $e_1$  lowEqual  $e_2$ ,

1002 1003 *then*:

983

984

985

995

 $\begin{array}{c} {}^{1003}\\ {}^{1004} \end{array} A) M_2 | e_2 \rightarrow_{call} M'_2 | e'_2 \ and \ e'_1 \ lowEqual \ e'_2, \end{array}$ 

B)  $M'_1$  similar $(e'_1) M'_2$ , 1005

C)  $M_1$  preserves $(e_1/e_1') M_1'$ ,

 $\begin{array}{c} \text{D} M_2 \ \text{preserves}(e_2/e_2') \ M_2', \\ 1007 \end{array}$ 

 $E) M'_{1}, M'_{2}, e'_{1}, e'_{2} are well typed$ 

PROOF. We prove that all five conditions A-E are satisfied. For A and B, we prove this theorem by cases on  $\rightarrow_{call}$ . We only prove the cases including a context, because the proofs with an empty context imply the correctness of the rules without a context.

## 1012 **Proof of A and B by cases:**

**1013** Case CTX + CALL:

1014 If  $e_1$  and  $e_2$  are of form  $\mathcal{E}_v[(high mdf o).m(\overline{o})]$ , the proof is by rule (HREC).

Proof of A: The only point of non-determinism in this language is the way new object identities are chosen, and the only way to introduce a new *o* is with the NEW rule. From assumption (1) and Lemma 5.8, we know that there is an execution of  $\rightarrow_{call}$ , containing an arbitrary number of reduction steps ( $\rightarrow$ ). The reduction is of form  $M|\mathcal{E}_v[(high mdf o).m(\bar{v})] \rightarrow_{call} M'|\mathcal{E}_v[(high mdf o'')]$ , where the result is a high value, thus by the definition of *lowEqual*,  $\mathcal{E}_v[(high \__)]$  *lowEqual*  $\mathcal{E}_v[high\__)]$ holds.

1021 Proof of B: We execute a number of steps on the high receiver. By assumption (1) and Lemma 5.8, 1022 this process terminates and produces a result. By typing rule CALL, we know the result is high 1023 and the parameters vs must only contain low read/imm/capsule values. By the assumption that the 1024 language satisfies the modifier properties (e.g. immutability), we do not modify the ROG of the low 1025 read/imm parameters. The low capsules are promoted to high in both memories, and thus, will not be 1026 part of the low ROG anymore.

All the other parameters and the receivers are high, so they will not influence the low values: the whole ROG of a high object is high, and computation can only influence reachable objects since we

do not have any static variables. So B holds since the (shrunken) low ROG on both memories cannotbe mutated.

<sup>1032</sup> If  $e_1$  and  $e_2$  are not of form  $\mathcal{E}_v[(high mdf o).m(\overline{o})]$ , the proof is by rule (HOTHER).

1033 Proof of A: In this case we are doing a single reduction step →. The only way to introduce non-1034 determinism is by creating a new object, but this step is a method call. Thus,  $M_1 = M'_1$ ,  $M_2 = M'_2$ 1035 and  $e'_1$  lowEqual  $e'_2$ .

Proof of B: The expressions  $e_1$  and  $e_2$  must be of form  $\mathcal{E}_v[(1 \text{ow } mdf \ o).m(\overline{o})]$  and the low receiver is the same on both sides. Moreover, since the original memories are similar, the low o is an instance of the same class, thus the method call will resolve to the same body, and the application of reduction rule (CALL) is deterministic and is identically in both cases, so the memories are not influenced (except for the usual shrinkage on promoted low capsule and low imm objects). Thus, B holds.

### **1042** Case CTX + FIELD UPDATE:

1043 *Proof of A:* By assumption (1), we know the expression reduces. The only way to introduce a 1044 new *o* is with the (NEW) rule; thus *A* holds since we use the same process to get  $v'_2$  (the rule applies 1045 a deterministic procedure).

1046 *Proof of B:* In this case, if  $v_2$  is low, we know:  $M_1 | \mathcal{E}_v [v_1.f_0 = v_2], M_2 | \mathcal{E}_v [v_1.f_0 = v_2]$ , and 1047  $M_1 similar(\mathcal{E}_v [v_1.f_0 = v_2]) M_2$ . Thus,  $M'_1 | v'_2, M'_2 | v'_2$ , and  $M'_1 similar(\mathcal{E}_v [v'_2]) M'_2$ .

If  $v_2$  is high, then we could have a different value in the second reduction, but this value will also be high, and thus, not influence similarity. By well typing, this high value will be stored in a high field. Intuitively, if  $v_1$  is low, we can assign a high only if  $f_0$  is high. In this case, the low ROG did not contain  $v_2$  and now does not contain  $v'_2$ , so *B* holds. If  $v_1$  is high, then it was not part of the low ROG to begin with, so *B* holds.

However, we need to inspect all possible field update cases to check that all assignments do notviolate our similarity property:

To shorten the writing for all cases, we assume a low reference low1, a high reference high1 with a low field lowF and a high field highF. The assigned objects can be low low2 or high high2.

• low1.lowF=low2 This is equal in both reduction, so *B* holds.

• low1.highF=low2 This is ok if low is imm or capsule. If we have removed the last low reference to 'low', then the result of \_[only \_] will shrink in the same way in both computations.

- high1.highF=low2 This is ok if low is imm or capsule, and thus, it is absent in the low ROG.
- high1.lowF=low2 This is ok if low is imm or capsule, and thus, it is absent in the low ROG.

Note for the capsule cases: The value  $v_2$  was low capsule in  $\mathcal{E}_v[v_1, f_0 = v_2]$ , thus it was kept as part of the low memory by \_[only \_]. In the next step  $v'_2$  is now high, thus it is not kept as part of the low memory by \_[only \_]. Since the only change between the result of the \_[only \_] memory is the absence of the low capsule, *B* holds.

- low1.lowF=high2 This is forbidden by the type system.
- low1.highF=high2 This is ok and irrelevant, since it does not change the low ROG.
- high1.highF=high2 This is ok and irrelevant, since it does not change the **low** ROG.
- high1.lowF=high2 This is ok and irrelevant, since it does not change the low ROG.

### Case CTX + FIELD ACCESS:

1072 Proof of A: The only way to introduce non-determinism is the way new objects are created. We 1073 are accessing a location o that is equal in both cases or it is a high location. As we are not modifying 1074 the memories,  $M_1 = M'_1$  and  $M_2 = M'_2$  and  $e'_1$  lowEqual  $e'_2$  holds.

1075 Proof of B: The memory is not changed by accessing a field.  $M_1 similar(e) M_2$  holds before and 1076  $M_1 = M'_1$  and  $M_2 = M'_2$ , thus B holds.

1077 1078

1070

1071

#### **1079 Case CTX + NEW:**

In this case, assumption (1) is of form  $M_1 | \mathcal{E}_v[\text{new } s \ C(\bar{v})] \rightarrow_{call} M_1, o \mapsto C(\_) | \mathcal{E}_v[(s \text{ mut } o)]$ , and (A) is similar with  $M_2$ .

1082 *Proof of A:* This can be obtained by simply choosing a suitable reference 'o' for the NEW rule.

*Proof of B*: It holds because the new memories grow by adding the same exact object on both sides.

1083 1084

1092

1108

## **1085 Proof of C and D:**

To prove the conditions C and D, we have to show that if M and e are well typed and  $M|e \rightarrow_{call} M'|e'$ then M preserves(e/e') M'. That means, each reduction step ensures the preserve property that the high memory is only changed in necessary cases. This statement holds by the construction of our reduction rules. We only promote expressions to high if it is necessary. No unnecessary promotions are done. The only changes from low to high are explicitly stated in Definition 5.7: creation of a new high object and the promotion of a low capsule expression. Thus, C and D holds.

#### 1093 Proof of E:

Proving condition *E* is more complex. From our assumptions we conclude that the well typedness 1094 of the base language is not violated, since the SIFO type system is just stricter than the regular L42 1095 1096 system. However, we need to prove that the  $\rightarrow_{call}$  reduction preserves the added security typing. This is quite subtle thanks to multiple method types (Fig. 3): The type system as presented does not 1097 respect preservation; that is, when calling a method that has been typed using multiple method 1098 types, the inlined body of the method may not respect our provided type system. However, we are 1099 using the  $\rightarrow_{call}$  reduction, and the call reduction skips in a single step to the full method evaluation. 1100 1101 The method body will evaluate to a value; we have not formally specified typing rules for values and memory; the intuition we present here in our proof sketch is that security is not relevant in 1102 the memory; as you can see from the grammar in Fig. 6, we keep the security level on the value 1103 (outside of the memory), so the result of a high method call with  $\rightarrow_{call}$  is well typed because it is 1104 only concerned with the non-security aspects of the type system, and the security level is tracked 1105 during reduction. See how, for example, in rule Call of Fig. 4, the security level and modifier of the 1106 1107 parameters can be promoted before inlining the method body. 

To prove the noninterference Theorem 5.10, we use the property that the reduction terminates 1109 (Lemma 5.8) and the bisimulation core that each reduction step meets the requirements of nonin-1110 terference (Lemma 5.9). We prove, if two memories that are similar and both expressions reduce, 1111 the new memories still have to be similar and the reduced values are equal w.r.t. low values. This 1112 Theorem 5.10 has the same shape as Theorem 5.0, but uses the definitions for similarity of memories 1113 (Def. 5.5) and expressions (Def. 5.6). As we said before, we exclude declassification in the proof. 1114 Therefore, we cannot guarantee security for programs with declassify expressions. Related works 1115 generalize the noninterference property to provide stronger guarantees for programs including 1116 declassification [Sabelfeld and Sands 2009]. 1117

#### THEOREM 5.10 (NONINTERFERENCE).

If we have expressions  $e_1$  and  $e_2$  without declassification that are well typed and  $e_1$  lowEqual  $e_2$ ,  $M_1$ and  $M_2$  are well typed memories,  $M_1$  similar $(e_1)$   $M_2$ ,  $M_1|e_1 \rightarrow^* M'_1|v_1$ , and  $M_2|e_2 \rightarrow^* M'_2|v_2$  then  $M'_1$  similar $(v_1)$   $M'_2$ ,  $v_1$  lowEqual  $v_2$ , and  $M'_1$ ,  $M'_2$ ,  $v_1$ , and  $v_2$  are well typed.

1123 PROOF. From Lemma 5.8 and  $M_1|e_1 \rightarrow M'_1|v_1$  and  $M_2|e_2 \rightarrow M'_2|v_2$ , we know that  $M_1|e_1 \rightarrow_{call}$ 1124  $M'_1|v_1$  and  $M_2|e_2 \rightarrow_{call} M'_2|v_2$ . Then, by induction on the number of steps of  $\rightarrow_{call}$ :

Base:  $e_1$  lowEqual  $e_2$  lowEqual  $v_1$  lowEqual  $v_2$ ,  $M_1 = M'_1$ ,  $M_2 = M'_2$ . Thus,  $M'_1$  similar( $v_1$ )  $M'_2$ holds because  $M_1$  similar( $e_1$ )  $M_2$ .

Inductive step: By Lemma 5.9 (bisimulation core) and the inductive hypothesis, each reduction step establishes similar memories  $M'_1$  and  $M'_2$ , computes *lowEqual* expressions, and preserves that only necessary values are in the high ROG.

1131 1132 6 TOOL SUPPORT AND EVALUATION

In this section we present tool support for SIFO and evaluate feasibility of SIFO by implementing
five case studies. Additionally, we benchmark precision and recall of the information flow analysis
by adapting the IFSpec benchmark suite [Hamann et al. 2018] to SIFO.

## 1137 6.1 Tool Support

We implement SIFO as a pluggable type system for L42 [Giannini et al. 2019]. L42 is a pure object-oriented language with a rich type system supporting the type modifiers used by SIFO.

Conveniently, L42 allows pluggable type systems [Andreae et al. 2006; Papi et al. 2008] to add an
additional layer of typing. We add rules to support the typing of expressions with security levels.
Both Java and L42 supports pluggable type systems using annotations: type names preceded by the
symbol @. In our SIFO library, these annotations are used to introduce the security levels.<sup>5</sup>

Some changes of SIFO are necessary to comply with L42: L42 supports the uniform access 1144 1145 principle [Meyer 1988]; thus there is no dedicated syntax for field assign and field access, but they are modeled by getters and setters. Additionally, the constructor does not have dedicated syntax, 1146 but it is a static method with return type This. In this way, we only need to type check method 1147 calls. Moreover, this allows more flexibility since multiple method types are now transparently 1148 and consistently applied in all those cases. We also had to extend our type system to support the 1149 1150 features of L42. While adding loops and other conventional constructs was trivial, we had to be careful while extending our type system to support exceptions, since exceptions constitute yet 1151 another way for a method execution to propagate secret information to an observer. Thus, we 1152 consider exceptions similar to a return type. 1153

Additionally, an exception prevents the execution of the code after it was thrown. Thus, after the exception is caught, the program may collect information about when the execution was interrupted in order to discover what expression raised the exception. This is another option to propagate secret information to an observer. Our current extension supporting exceptions is quite conservative, requiring the use of a single security level for all free variables, exceptions, and results of a try-catch block. In future work, we plan to formalize the extension with exceptions more precisely.

1160

1166

## **1161 6.2 Feasibility Evaluation**

To evaluate the feasibility of SIFO, we implemented four case studies from the literature in SIFO:
Battleship [Stoughton et al. 2014; Zheng et al. 2003], Email [Hall 2005], Banking [Thüm et al. 2012],
and Paycard (http://spl2go.cs.ovgu.de/projects/57). Additionally, we implemented a novel
case study of our own, the *Database*. The metrics of the case studies are shown in Table 1.

6.2.1 Battleship. Our evaluation is focused on the Battleship case study because this program is carefully described by Stoughton et al. [Stoughton et al. 2014] as a general benchmark to evaluate information flow control. Moreover, this case study is also implemented in Jif<sup>6</sup>, thus allowing us to directly compare their results with our work.

*Battleship* is the implementation of a two player board game. At the start, each player places a fixed number of ships of varying length on their private board. The board has a two-dimensional

 $<sup>\</sup>overline{}^{5}$ You can find a version of L42 with our SIFO library and the case studies at https://l42.is/SifoArtifactLinux.zip and https:

<sup>&</sup>lt;sup>1174</sup> //l42.is/SifoArtifactWin.zip. This also contains more information about the detailed syntax in the readme.

<sup>&</sup>lt;sup>1175</sup> <sup>6</sup>See [Zheng et al. 2003] found on the Jif website https://www.cs.cornell.edu/jif/

<sup>1176</sup> 

1187 1188 1189

Name	#Security Levels	#Security Annotations	#Lines of Code	
Battleship4 (+2 generic)Database4Email2		21 (208 in Jif)	431 (611 in Jif) 73 260	
		6		
		20		
Banking	2	20	127	
Paycard	2	15	95	

Table 1. Metrics of the case studies

grid. The players only know the placement on their board and have to guess where the other player placed the ships. During the game, the players take turns and *shoot* cells on the board of the opponent to sink ships. The first player wins the game who sinks all opponent's ships.

Thanks to our flexible SIFO type system, we implemented most of the code without any security annotations. We wrote a generic PlayerTrait trait that is parameterized over the security levels *SelfL* and *OtherL* to distinguish both players. Our implementation of *Battleship* uses many features from full L42, not just the minimal core presented in this paper. In particular, we rely on L42 traits and their encoding for generics. We will expand on this in Section 7.

Our ExampleGame class implements a mutually distrustful player scenario [Stoughton et al. 2014]. 1198 In this setting, even if one of the two players is replaced with an adversarial player, we ensure that 1199 only a correctly executed game will terminate without exception. This scenario highlights nicely 1200 the properties of our system: we ensure noninterference that an adversarial player is unable to read 1201 the opponent's board state. In ExampleGame, the two players Player1 and Player2 create the boards 1202 and shoot consecutively. We have to ensure that each player creates a confidential board that the 1203 other player cannot read. In the concrete implementation, we annotate the board with the security 1204 level of one of the players to restrict readability. Deep immutability enforces that the board is not 1205 manipulated during the game. Then, each player gets a reference to the confidential board of the 1206 opponent. Contrast this with the Jif implementation of the same game: Jif uses the concept of label 1207 expressions that specify the allowed readers and writers of objects. In Jif, boards can be read by 1208 only one player, but they are trusted by both players. The first player creates a board, the second 1209 player endorses this board, and the first player then saves this board that is trusted by both players. 1210 Endorsement is the name of downgrading integrity, similar to declassification for confidential data. 1211 The endorsement of the board is implemented in Jif with defensive cloning. A new trusted board is 1212 created, and all ships on the input board are cloned to add them to the trusted board. In SIFO, the 1213 endorsement and defensive cloning is not necessary because we can rely on deep immutability of 1214 the type system that prevents manipulation of the boards. 1215

When a player shoots, it has to be correctly revealed if it was a miss, a hit, or a hit that sunk a 1216 specific ship. The process of one shooting round is shown in Listing 4. The method round has as 1217 parameter which is the opposing player. In Line 2, there is a dynamic check for validity of the game. 1218 x works like assert in Java. It must be the round of the player, and the game must not be yet won 1219 by any player. The method fire in Line 3 asks for the next coordinates to shoot. These coordinates 1220 are used to check for the result of the shot on the board. As this is the board of the opponent, 1221 the result is labeled with the opponent's security level (Line 4). We now have to declassify the 1222 result to be able to read it, but only the opponent has the right to declassify the result of such 1223 shot. An adversarial opponent could manipulate the declassified result, we therefore use a signing 1224 1225

```
12261
      mut method Bool round(mut OtherPlayer other) = (
12272
        X[this.myRound(); !this.win(this.myShots()); !this.win(this.otherShots())]
12283
        coord = this.fire()
        @OtherL FireResult res = this.board().fire(coord=coord,shots=this.myShots())
12294
        ResultSigner s = this.signer()
12305
        @OtherL ResultSigner.Signed signed = s(label=coord.toS(),data=res)
12316
        ResultSigner.Signed freeSigned=other.declassify(signed)
12327
1233
9
        X[this.signer().mine(freeSigned,label=coord.toS())]
        r = freeSigned.data().answer().repr()
1234
10
        this.myShots(\myShots.with(row=coord.row(),col=coord.col(),val=r))
<sup>1235</sup>11
        this.myRound(Bool.false())
12362
        this.win(this.myShots())
12373
     )
1238
```

Listing 4. Implementation of one shooting in SIFO

mechanism to exclude manipulation. Thus, the confidential result is signed by the shooting player and send to the opponent (Line 6), so that the opponent declassifies the result (miss, hit, ship sunk) in Line 7. In Line 8, it is checked that the correct signed result is returned. In our implementation, we can rule out manipulation as the correct result is immutable and a newly created result by the opponent cannot have the signature of the shooting player. The shot and the result are then added to a list for validating subsequent rounds (e.g., in Line 12 to check whether the game is won). Then, the opponent player takes turn. If a game rule violation is detected during the game (e.g., a manipulated result of a shot), the game can be aborted by either player.

In Jif, a player must trust that the result of a shot is correctly revealed by the opponent. In Jif, it would be easy to implement a BadPlayer that returns an incorrect result of a shot, as there is no check for a manipulation [Stoughton et al. 2014]. Additionally, the Jif implementation uses defensive cloning when passing the coordinates of a shot to the opposing player.

1253 We now compare both implementation on a more general level. Most classes are written parameterized with a security level L in Jif. In SIFO, we write classes like Ship without any security annotation, but we are able to use them in secure contexts with our promotion rules. With this technique, we are able to write only 21 security annotations in the whole implementation. In Jif, we count 208 annotations.

1258 For the creation of players, we have a similar concept as Jif. While Jif used a generic Player class, 1259 we created a generic PlayerTrait trait with two security levels *SelfL* and *OtherL*. This generic trait is 1260 instantiated as PlayerTrait['SelfL=>Player1; 'OtherL=>Player2] which can then be used to create object 1261 instances. In Jif, [Player1, Player2] player1 = new Player[Player1, Player2](); is written to create a new 1262 player. 1263

In summary, we implemented *Battleship* in SIFO by relying on our promotion rules and the immutability of trustworthy objects. Thanks to preexisting L42 features, we have not needed the complex expressiveness of Jifs label expression, which is also discussed in Section 7.

6.2.2 Database. The Database is a system where two databases are not allowed to interfere. 1267 Through the different security levels, we ensure that a value read from one database cannot be 1268 inserted into the other one. This can be obtained just by annotating the Gui class with six security 1269 annotations, as shown in Listing 5. 1270

```
12711
      Gui={
12722
        mut @Left Database dbLeft
        mut @Right Database dbRight
12733
1274
```

ACM Trans. Program. Lang. Syst., Vol. 1, No. 1, Article 1. Publication date: January 2022.

1239 1240 1241

1242

1243

1244

1245

1246

1247

1248

1249

1250

1251

1252

1254

1255

1256

1257

1264

1265

```
class method mut This (mut @Left Database dbLeft, mut @Right Database dbRight)
12754
        class method mut This ()=(
12765
          capsule @Left Database dbl=Database(name=S"left",rows=Rows())
12776
          capsule @Right Database dbr=Database(name=S"right",rows=Rows())
12787
          This(dbLeft=dbl, dbRight=dbr)
1279<sup>8</sup>
12809
          )
10
1281
        }
                                    Listing 5. Gui implementation in SIFO
1282
1283
```

The other classes are implemented without any security level. This is possible because the class Gui is the only class that uses databases with different security levels. This means that the actual database code is all free from security annotations; as you can see above different instances of low capsule Database can be transparently promoted to different security levels Left and Right.

6.2.3 Further case studies. The Email system ensures that encrypted emails are only decrypted if
the public and private key pair used is valid. It also guarantees that private keys are not leaked.
The Email system needed only 20 security annotations in 260 lines of code.

Banking and Paycard are two systems that represent payment systems where it is crucial that the calculations of new balances are correct and information is not leaked. By setting the balance to high and checking that the system is typable, we are certain that the balance is not leaked to attackers. For BankAccount and Paycard 20 and 15 security annotations were needed with 127 and 95 lines of code.

6.2.4 Discussion. Code following a pure object-oriented style is often directly supported by SIFO 1298 without any special adaptation. However, when updates to local variables and statements/condi-1299 tionals are used, the programmer may have to rely on some simple programming patterns: for 1300 example, in a conditional, we cannot directly update a high field of a low object, as low objects cannot 1301 be manipulated in high conditional statements. This limitation can be circumvented by wrapping 1302 the updatable field into a mutable proxy object (e.g., o.proxy.field instead of o.field). Such a proxy 1303 object can then be saved in a high temporary variable; and such a variable can be used to manipulate 1304 the state. We have to use the high proxy object because our IF-rule is slightly conservative. It does 1305 not check if only high fields of a low object are manipulated. This is still easier in comparison to 1306 Jif because in Jif, the object instance has to be cloned so that the user keeps a reference to the low 1307 object and can manipulate the high field data with the cloned instance. 1308

Another insight is that major parts of the case studies could be written without any use of 1309 security levels because the multiple method types promote the parameters of a called method to 1310 the required security levels in necessary cases. This allowed us to write secure programs relying 1311 on libraries and data structures without any security annotation. In our case study, we could reuse 1312 a list implementation and securely promote it to any security level if needed. The type system 1313 then checks that instantiated lists of different security levels did not interfere. For example, a 1314 high list can only contain objects of at least a high security level. Moreover, we were also able to 1315 encode domain-specific data structures and functionality without any security annotation. In Jif, 1316 this requires generic classes written with security annotations in all cases. Information flow can be 1317 enforced by annotating just the few method bodies that put separate systems into communication. 1318

Major parts of the case studies were implemented without the use of declassify. We only needed it in *Battleship* to declassify shot results as intended by the game. Additionally, we declassified console output at the end of program execution in the other case studies to print results for the user.

1284

1285

1286

1287

1288

By explicitly typing references as imm or capsule, the code quality increases, because programmers can rely on properties which are enforced by the type system. Furthermore, the security levels serve as active documentation for the programmer. In the *Database* example, we know which database a value comes from by reading the security level. To reduce the writing effort for programmers, sensible defaults are useful: if a security level is not specified, **10**w is used.

## 1330 6.3 Benchmarking with IFSpec

To evaluate precision and recall of SIFO, we applied SIFO to the IFSpec benchmark suite [Hamann 1331 et al. 2018]. IFSpec contains 80 samples to test information flow analysis tools. In addition to 1332 the core samples, 152 samples from the benchmark suite SecuriBench Micro<sup>7</sup> are adapted and 1333 integrated into IFSpec. The samples are all available in Java and Dalvik. To benchmark SIFO, 1334 we translated the 80 core sample when it was possible. Samples that used Java specific features 1335 1336 were not translated. In total 40 samples are implemented in SIFO. For these samples, we compare SIFO with Cassandra [Lortz et al. 2014], JOANA [Graf et al. 2013], JoDroid [Mohr et al. 2015], 1337 KeY [Ahrendt et al. 2016], and Co-Inflow [Xiang and Chong 2021] (with and without additional 1338 security annotations) which were all evaluated before with IFSpec. 1339

Each sample is labeled as secure or insecure. When a sample contains a leak and a tool reports a 1340 1341 leak, we categorize it as true positive (TP). When a sample contains no leak and a tool reports no leak, we categorize it as true negative (TN). When a sample contains no leak but a tool reports a 1342 leak, we categorize it as false positive (FP). When a sample contains a leak but a tool reports no leak, 1343 we categorize it as false negative (FN). From these four categories, we can calculate precision and 1344 recall of the tools. The recall is computed as: #TP/(#TP + #FN). Recall determines the percentage of 1345 samples correctly classified as insecure considering all samples containing a leak. The precision is 1346 computed as: #TP/(#TP + #FP). Precision determines the percentage of samples correctly classified 1347 as insecure considering all samples classified as insecure by the tool. 1348

In Table 2, we show the results of the benchmarking. All six tools found the 18 samples containing
a leak. This results in a recall of 100% for all tools. No tool classified a sample false negative. Regarding
precision, the tools have slight differences. Cassandra and Co-Inflow without additional annotations
have the lowest precision of 54.5%. JOANA has the highest precision of 62.1%, but Co-Inflow has a
higher precision of 81.8% if additional security annotation is given by the programmer. SIFO has a
precision of 58.1%.

6.3.1 Discussion of the False Positive Samples. With SIFO, 13 samples are typed as insecure, which
are labeled as secure by the authors of the benchmark. We will classify these samples into categories
to discuss the result of SIFO. For six samples, the type system of SIFO is not precises enough to
recognize that the sample is secure. For example, if in a conditional expression both branches assign
the same value to a low reference, we pessimistically dismiss this program.

Three samples are constructed to introduce a leak which is overwritten in the end. A simple example is that a secret value is assigned to a public variable and in the next line, the public variable is overwritten. We clearly prohibit the first assignment of the secret value. These examples are constructed for taint analysis tools and are not suitable for type systems.

One sample is labeled as secure because in the provided code there is no way to access the secret values. In sample webstore, a secret and a public value are assigned to a public list and only the public value is accessed in the code. When a simple getter-method for the secret value is added, the sample would be insecure. As our type system is modular, we directly prohibit the assignment of secret values to the public list. We do not check that there is currently no available method to access the stored value.

1361

1362

1363

1364

<sup>1371 &</sup>lt;sup>7</sup>https://github.com/too4words/securibench-micro

Tool	#Samples	ТР	TN	FP	FN	Recall	Precision
Cassandra	40	18	7	15	0	100%	54.5%
JOANA	40	18	11	11	0	100%	62.1%
JoDroid	40	18	9	13	0	100%	58.1%
KeY	40	18	8	14	0	100%	56.3%
Co-Inflow	40	18	7	15	0	100%	54.5%
Co-Inflow- Annotations	40	18	18	4	0	100%	81.8%
SIFO	40	18	9	13	0	100%	58.1%

Table 2. Overview of the benchmark results

For three samples, again the modular reasoning of the type system pessimistically rejects secure programs. In DeepCall, a chain of method calls is insecure because the first secret value is propagated through all calls and returned as a public value. In the similar sample DeepCall2, the last call always returns the same value independent of the secret input value. This sample is considered secure. Our types system does not check all method calls globally, it just reasons that a secret value is passed to the next method and that a secret return value is expected. That the secret return value is actually a public constant in the sample DeepCall2 is outside of the modular reasoning.

Most examples that SIFO rejects are constructed by developers to contain a security problem which is erased or not accessible in the remaining code of the sample. As our type system prohibits any introduction of security violations, we reject these samples. To support this statement, we rewrote eight of the 13 false positive samples to be semantically similar and accepted by SIFO.

## 7 RELATED WORK

1390

1391

1392

1393

1394

1395

1396

1397

1398

1399

1400 1401

Static and dynamic program analysis [Austin and Flanagan 2009; Nielson et al. 1999; Russo and
Sabelfeld 2010; Zhang et al. 2015], as well as security type systems [Banerjee and Naumann 2002;
Ferraiuolo et al. 2017; Hunt and Sands 2006; Li and Zhang 2017; Simonet 2003; Volpano et al. 1996]
are used to enforce information flow policies. We refer to Sabelfeld and Myers [2003] for a detailed
overview.

Taint Analysis. Taint analysis [Arzt et al. 2014; Enck et al. 2014; Hedin et al. 2014; Huang et al. 1408 2014, 2012; Milanova and Huang 2013; Roy et al. 2009] is a related analysis technique that detects 1409 direct information flows from tainted sources to secure sinks by analyzing the assignments of 1410 variables and fields. Those taint analysis works do not provide a soundness property, while the 1411 SIFO noninterference proof guarantees the security of type checked code. Except for JSFlow [Hedin 1412 et al. 2014], Cassandra [Lortz et al. 2014], JOANA [Graf et al. 2013], and JoDroid [Mohr et al. 2015], 1413 these related works do not cover implicit information flows through conditionals, loop statements, 1414 or dynamic dispatch. SIFO also detects implicit information flows through dynamic dispatch 1415 (conditional and loop statements are not in the core language, but included in our implementation). 1416 Crucially, the noninterference proof of SIFO relies on detecting implicit information flow. 1417

Coarse-grained dynamic information flow approaches [Jia et al. 2013; Nadkarni et al. 2016; Xiang and Chong 2021] track information at the granularity of lexically or dynamically scoped section of code. Instead of labeling every value individually, coarse-grained approaches label an

entire section with one label. All produced values within this scope implicitly have that same label. 1422 Therefore, the writing effort for developers to annotate programs is reduced. To still obtain good 1423 1424 results of the information flow analysis, for example, Xiang and Chong [Xiang and Chong 2021] introduce opaque labeled values to permit labeled values where programmers have not provided a 1425 label. If no further annotation is given by the programmer, the precision of the information flow 1426 analysis can be decreased. As the evaluation shows, Co-Inflow [Xiang and Chong 2021] has better 1427 precision when the programmer annotates the program. However, the precision is not a limitation 1428 1429 of coarse-grained approaches compared to fine-grained approaches. Type systems for fine- and coarse-grained information flow control are equivalent in terms of precision as shown by Rajani et 1430 al. [Rajani et al. 2017; Rajani and Garg 2018]. For dynamic information flow control mechanisms, 1431 Vassena et al. [Vassena et al. 2019] have similar results. 1432

The work by Huang, Milanova et al. [Huang et al. 2014, 2012; Milanova and Huang 2013] is closely related to our approach because viewpoint adaption with polymorphic types is similar to our mdf > mdf' operator for type modifiers. For field accesses, the type of the accessed object depends on the reference and the field type. They use read-only references to improve the precision of their static analysis technique by allowing subtyping if the reference is read-only. In SIFO, we also use deep immutable and capsule references, extending the expressiveness of our language.

Comparison to Jif. In this work, we explored the specific area of secure type systems for object-1440 oriented languages [Banerjee and Naumann 2002; Barthe et al. 2007; Barthe and Serpette 1999; 1441 Myers 1999; Sabelfeld and Myers 2003; Strecker 2003; Sun et al. 2004]. The most important work 1442 to compare against is Jif [Myers 1999] (see Section 2). In this paper, we presented a minimal core 1443 of SIFO for the soundness and noninterference proofs. Nonetheless, we compare SIFO with Jif by 1444 discussing their common and different features. A main difference is the handling of aliases: Jif 1445 does not use any kinds of regions or alias analysis to reason about bounded side effects. Therefore, 1446 Jif pessimistically discards many programs introducing aliases (see the example in Section 2.1 that 1447 is not typable in Jif). On the other hand, SIFO restricts the introduction of insecure aliases and is 1448 therefore able to safely type more programs. SIFO's expressiveness relies on the safe promotion of 1449 imm or capsule references. As shown in Section 2 and in Section 6.2, programs can be typed securely 1450 without defensive cloning [Bloch 2016] because imm and capsule modifiers allow promoting objects 1451 to higher security levels. In Jif, a similar promotion is only allowed for primitive types. 1452

The SIFO type system leverages on a minimalistic syntax of security annotation, where types contain a security level. Jif offers a much more elaborated syntax: in Jif, a security label is an expression consisting of a set of policies [Myers and Liskov 2000]. Each policy has an owner oand a set of readers r. For example, the label  $o_1 : r_1; o_2 : r_1, r_2$  states that the policy of  $o_1$  allows  $r_1$ to read the value and  $o_2$  allows  $r_1$  and  $r_2$  to read the value. Hence,  $r_1$  is the only reader to fulfill both policies. These label expressions get more complicated, the more policies are conjoined, but a programmer gets more flexibility to express fine-grained access restrictions.

SIFO has a similar expressiveness to Jif, but does not need to resort to such complex label 1460 expressions. To show this, consider the following scenario from Jif [Myers and Liskov 2000]: 1461 A person Bob that wants to create his tax form using an online service. In the scenario, Bob 1462 wants to prevent his information from being leaked to the online service, and the provider of the 1463 service does not want its technology and data to be leaked in the process of generating the tax 1464 form. This constraint is related to the mutually distrustful players of the Battleship case study. To 1465 comply with these constraints, Bob labels his data with *bob* : *bob* and sends it to the online service 1466 provider. The provider, labels its own data with provider : provider, so the calculated tax has the 1467 label *bob* : *bob*; *provider* : *provider*. This result cannot be read because the labels disagree on their 1468 reader sets. To release the information to Bob, the provider declassifies the label by removing the 1469

1470

```
14711
      class Protected {
14722
         final label{this} lb;
14733
         Object{*lb} content;
         public Protected{LL}(Object{*LL} x, label LL) {
14744
           lb = LL;
14755
1476<sup>6</sup>
           super();
           content = x;}
1477<sup>7</sup>
         public Object{*L} get(label L):{L} throws (IllegalAccess) {
۲478
9
   8
           switch label(content) {
1479
10
              case (Object{*L} unwrapped) return unwrapped;
1480
11
              else throw new IllegalAccess();}}
1481_{12}
         public label get_label() {
14823
           return lb;}}
1483
                     Listing 6. Class Protected in Jif with security parameterization [Myers 1999]
1484
```

provider policy. As only the final tax form is declassified, the released information from the provider
is limited. The final tax form with the label *bob* : *bob* is then sent to Bob.

In SIFO, we can handle the same scenario as follows: Bob wants to protect his private information, 1490 so he can set the security level to **bob**, but he can also set a type modifier. With read, he ensures 1491 that his data cannot be manipulated and integrated into the provider's data. With imm, only the 1492 manipulation is prevented. If Bob trust the provider, he can send a capsule object to the provider. 1493 The provider can then manipulate and alias the data, but Bob is sure, that the manipulation is 1494 restricted to only the data reachable from the given reference. In the case of the provider, they get 1495 a reference to the data of Bob with a security level and a type modifier. In the most restricted case 1496 of a read reference, the provider can still use the information and calculate the final tax form, but 1497 a manipulation or aliasing of Bob's data is prevented. The security level of the result is the least 1498 upper bound of **bob** and **provider**. To declassify the results for Bob, the final tax form needs to be imm 1499 or capsule to allow safe sharing or transfer of the confidential data. 1500

With this example, we discuss the secure transfer of data. In SIFO, by using a read, imm, or capsule modifiers, Bob specifies how the information is usable. In Jif, the label bob:bob does not restrict the use in the same way. There is no language support to ensure that a unique portion of store is transferred to the provider. There is also no guarantee that the data is not manipulated, as with imm. In Jif, if the provider has a read permission for Bob's data, they can freely manipulate it. Furthermore, if the provider wants to ensure that they are the current owner of Bob's tax data, they have to clone the data (defensive cloning [Bloch 2016]).

To grasp the difference in the annotation burden, consider the Jif example in Listing 6 of a class that protects data from insecure access. Jif uses a parameterized label system where a class or methods have a generic label L. The label L can be initialized with any specific security level. This places a large conceptual burden on the programmer, as the label L is used in every field and method of the class. Additionally, no legacy code can be used that is not parameterized properly.

SIFO encourages a style where most code is completely clear of any security annotation; in particular, most algorithms and most common data types like collections does not need any kind of security annotations at all. Only code that is explicitly and directly involved in the handling of security-critical data needs to be written with security in mind. Unlabeled classes and methods are implicitly annotated with the lowest security level. Thanks to the flexibility of multiple method types, they can be safely promoted to any higher level.

Jif has additional features that are not presented in the core of SIFO. The SIFO core works with a finite lattice of security levels instead of the complex label expressions in Jif [Myers and Liskov 2000] with an infinite set of possible labels. Thanks to the embedding in L42, we get label polymorphism for free by relying on L42 encodings for generics. Thus, on one side SIFO allows to remove the complexity of having most of the labels generic, on the other side when generic labels are truly needed (for example to write code that have to work on unknown labels) we can rely on the L42 generics encoding, as we do in the BattleShip example.

Jif has dynamic checks of security labels. See Line 9 in Listing 6 where the security level of the object content is checked. This feature can be emulated in SIFO with the following programming pattern. Any is the equivalent of Object in Java.

```
15301 BoxLeft=Data:{@Left Any left}
15312 BoxRight=Data:{@Right Any right}
15323 ...
15334 low Any a
15345 if a <:BoxRight return a.right()</pre>
```

In a more concrete example, a Person Bob creates a BoxLeft or BoxRight object with the secure data in the field. This object is then sent as **low** Any a to Alice and Alice can discover with **instanceof** (<: in L42) if it is a BoxLeft or a BoxRight object. As you can see, by knowing the explicit type, we know also the security level of the data in the field. Thus, by adding an explicit boxing step, we enable the users to handle any kind of label and to dynamically check on those.

Jif has robust declassification [Chong and Myers 2006] which means that an attacker is not able to declassify information, or to influence what information is declassified by the system that is above the security level that the attacker is allowed to read. In the full embedding in L42, declassification can be sealed behind the object capability model, as we did in the *Battleship* case study. The L42 object capability model is flexible and can provide a range of useful guarantees [Miller 2006]. Indeed, you can see the *Battleship* case study as an exemplar representation of robust declassification. Even if we replace one of the player with adversarial code, such code will not be able to declassify the opposing board; even while holding a reference to such a board.

In future work, we want to extend SIFO to work with any partial order of security levels as discussed in Section 9. With this feature, we are closer to the expressive power of Jifs label expressions.

Other Information Flow Techniques. Hoare-style program logics are also used to reason about 1552 information flow. The work of Andrews and Reitman [Andrews and Reitman 1980] encodes infor-1553 mation flow in a logical form for parallel programs. Amtoft et al. [Amtoft et al. 2006; Amtoft and 1554 Banerjee 2004] use Hoare-style program logic and abstract interpretation to analyze information 1555 flow. This approach is the basis in SPARK Ada for specifying and checking information flow [Amtoft 1556 et al. 2008]. For Java, Beckert et al. [Beckert et al. 2013] formalized the information flow property in 1557 a programming logic using self-composition of programs and an existing program verification tool 1558 to check information flow. Similarly, Barthe et al. [Barthe et al. 2004] and Darvas et al. [Darvas et al. 1559 2005] analyze the information flow by using self-composition of programs and standard software 1560 verification systems. Terauchi and Aiken [Terauchi and Aiken 2005] combined a self-composition 1561 technique with a type system to profit from both techniques. Küsters et al. [Küsters et al. 2015] 1562 propose a hybrid approach by using JOANA [Graf et al. 2013] and verification with KeY [Ahrendt 1563 et al. 2016] to check the information flow. 1564

The related IFbC approach by Schaefer et al. [Runge et al. 2020; Schaefer et al. 2018] ensures
information flow-by-construction. Here, the information flow policy is ensured by applying a sound
set of refinement rules to a starting specification. Instead of checking the security after program

1527

1528

1529

1535

1536

1537

1538

1539

1540

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

creation, the programmer is guided by the rules to never violate the policy. Compared to SIFO, theirapproach is limited to a while language without objects.

## <sup>1572</sup> 8 CONCLUSION

1571

1585

1586

1617

1573 In this work, we presented a type system of an object-oriented language for secure lattice-based 1574 information flow control using type modifiers that detects direct and implicit information flows. 1575 This language supports secure software development by enforcing noninterference. We leverage 1576 previous work on immutability and encapsulation to greatly increase the expressive power of our 1577 language. Additionally, promotion/multiple method types encourage reusability of secure programs 1578 without burdening the developer. We formalized the secure type system, proved noninterference, 1579 and showed feasibility by implementing SIFO as a pluggable type system for L42, and conducting 1580 an evaluation with several case studies. In the future, we want to formalize exceptions in SIFO 1581 to extend the expressiveness of the language. We also want to generalize the proof to include 1582 declassification.Furthermore, we could reduce the typing effort of programmers by introducing 1583 type inference. 1584

## 9 FUTURE WORK: INTEGRITY AND CONFIDENTIALITY

As noted by Biba [Biba 1977] integrity can be seen as a dual to confidentiality, which means 1587 that either of them can be checked with the same information flow analysis techniques. For 1588 confidentiality, information must not flow to inappropriate destinations; dually, for integrity, 1589 information must not flow from inappropriate sources. In this work, we made all our discussion 1590 about confidentiality. If a user of SIFO is instead interested in integrity, they can simply use our 1591 type system with any lattice of integrity levels. However, it is also possible to track both properties 1592 at the same time: The trick is to not rely too much on data sources with the lowest or highest 1593 security level (e.g. low and high): since high can see all the information, high offers no integrity. In 1594 the same way, low can write to all the information, thus low data needs to be always intrinsically 1595 valid/trusted. Note that we can still declare low fields and low data structures, it is sufficient for 1596 sensitive data to be stored somewere nested inside the ROG from a non low reference, as we shown 1597 with the database case study. 1598

In this work, we assumed a lattice of security levels. 1599 However, Logrippo [Logrippo 2018] has proposed that 1600 just a partially ordered set would be appropriate to model 1601 security. If we allowed just a partial order of security lev-1602 els, SIFO would allow to encode both integrity and con-1603 fidentiality at the same time instead of using two lattices 1604 for confidentiality levels and integrity levels. Consider 1605 the following example, where Bob and Alice have both 1606 confidential and trusted data. We can define a partially 1607 ordered set as shown on the right. 1608

Integrity: aliceTrusted/bobTrusted is data that Al ice/Bob trust to be valid. For example, only low and
 bobTrusted can write on bobTrusted data. Confidentiality:
 aliceConfidential/bobConfidential is data that Alice/Bob



wants to keep private. For example, low, aliceTrusted, bobTrusted, and bobConfidential can write on
 bobConfidential, but bobConfidential can only be read by bobConfidential and high. Those security
 levels also imply that bobTrusted can be read by bobTrusted, aliceConfidential, bobConfidential, and
 high.

1:34

1621

Being able to express integrity and confidentiality at the same time with the same lattice is clearly a great advantage; however we are still investigating if supporting partially ordered sets instead of a lattice would have subtle consequences that interfere with our noninterference property.

### 1622 REFERENCES

- Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016.
   Deductive Software Verification The KeY Book From Theory to Practice. Lecture Notes in Computer Science, Vol. 10001.
   Springer.
- Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. In POPL. 91–102.
- Torben Amtoft and Anindya Banerjee. 2004. Information Flow Analysis in Logical Form. In SAS (LNCS, Vol. 3148). Springer, 100–115.
- Torben Amtoft, John Hatcliff, Edwin Rodríguez, Robby, Jonathan Hoag, and David A. Greve. 2008. Specification and
   Checking of Software Contracts for Conditional Information Flow. In *FM*. Springer, 229–245.
- Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. 2006. A Framework for Implementing Pluggable Type Systems. In *OOPSLA*. 57–74.
- Gregory R. Andrews and Richard P. Reitman. 1980. An Axiomatic Approach to Information Flow in Programs. *TOPLAS* 2, 1 (1980), 56–76.
- Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien
   Octeau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware
   Taint Analysis for Android Apps. In *PLDI*, Vol. 49. ACM, 259–269.
- Thomas H Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *PLAS*. ACM, 113–124.
- Anindya Banerjee and David A Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language..
   In *CSFW*, Vol. 2. 253.
- 1640 Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In CSF. IEEE, 100–114.
- Gilles Barthe, David Pichardie, and Tamara Rezk. 2007. A Certified Lightweight Non-Interference Java Bytecode Verifier. In European Symposium on Programming. Springer, 125–140.
- Gilles Barthe and Bernard P Serpette. 1999. Partial Evaluation and Non-Interference for Object Calculi. In *FLOPS*, Vol. LNCS.
   Springer, 53–67.
- Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H Schmitt, and Mattias Ulbrich. 2013.
   Information Flow in Object-Oriented Software. In *LOPSTR*, Vol. LNCS. Springer, 19–37.
- D Elliott Bell and Leonard J La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report. MITRE Corp Bedford MA.
- 1648 Kenneth J Biba. 1977. Integrity Considerations for Secure Computer Systems. Technical Report. MITRE Corp Bedford MA.
- 1649 Joshua Bloch. 2016. *Effective Java*. Pearson Education India.
- 1650
   Stephen Chong and Andrew C Myers. 2006. Decentralized Robustness. In 19th IEEE Computer Security Foundations Workshop

   1651
   (CSFW'06). IEEE, 12-pp.
- Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *SPC*, Vol. LNCS. Springer, 193–209.
- <sup>1653</sup> Dorothy E Denning. 1976. A Lattice Model of Secure Information Flow. *CACM* 19, 5 (1976), 236–243.
- William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick
   McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring
   on Smartphones. *TOCS* 32, 2, Article 5 (June 2014), 29 pages.
- Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. 2017. Secure Information Flow Verification with Mutable Dependent Types. In *DAC*. IEEE, 1–6.
- Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible Recovery of Uniqueness and Immutability.
   *Theoretical Computer Science* 764 (2019), 145–172.
- 1660 Joseph A Goguen and José Meseguer. 1982. Security Policies and Security Models. In S&P. IEEE, 11-11.
- Adele Goldberg and David Robson. 1983. Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman
   Publishing Co., Inc.
- Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference
   Immutability for Safe Parallelism. 47, 10 (2012), 21–40. https://doi.org/10.1145/2398857.2384619
- Jürgen Graf, Martin Hecker, and Martin Mohr. 2013. Using JOANA for Information Flow Control in Java Programs A
   Practical Guide. In Proceedings of the 6th Working Conference on Programming Languages (ATPS'13) (Lecture Notes in

<sup>1666</sup> 

1667 Informatics (LNI) 215). Springer, 123–138.

- 1668 Robert J Hall. 2005. Fundamental Nonmodularity in Electronic Mail. Automated Software Engineering 12, 1 (2005), 41–79.
- Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. 2018. A Uniform Information Flow Security Benchmark Suite for Source Code and Bytecode. In *Nordic Conference on Secure IT Systems*. Springer, 437–453.
- Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript
   and Its APIs. In SAC (Gyeongju, Republic of Korea). ACM, 1663–1671.
- Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-based Taint Analysis for Java Web Applications. In *FASE (LNCS, Vol. 8411)*. Springer, 140–154.
- Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. *ACM SIGPLAN Notices* 47, 10 (2012), 879–896.
- 1676 Sebastian Hunt and David Sands. 2006. On Flow-Sensitive Security Types. SIGPLAN Not. 41, 1 (Jan. 2006), 79-90.
- Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ.
   *TOPLAS* 23, 3 (2001), 396–450.
- Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and
   Yutaka Miyake. 2013. Run-time Enforcement of Information-Flow Properties on Android. In *European Symposium on Research in Computer Security*. Springer, 775–792.
- Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid
   Approach for Proving Noninterference of Java Programs. In 2015 IEEE 28th Computer Security Foundations Symposium.
   IEEE, 305–319.
- 1684 Peixuan Li and Danfeng Zhang. 2017. Towards a Flow-and Path-Sensitive Information Flow Analysis. In CSF. IEEE, 53–67.
- Luigi Logrippo. 2018. Multi-level Access Control, Directed Graphs and Partial Orders in Flow Control for Data Secrecy and Privacy. In *Foundations and Practice of Security*. Springer International Publishing, 111–123.
- Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. 2014. Cassandra: Towards
   a Certifying App Store for Android. In Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &
   Mobile Devices. 93–104.
- Bertrand Meyer. 1988. Eiffel: A Language and Environment for Software Engineering. Journal of Systems and Software 8, 3 (1988), 199–246.
- Ana Milanova and Wei Huang. 2013. Composing Polymorphic Information Flow Systems with Reference Immutability. In
   *FTfJP*. ACM, Article 5, 7 pages.
- Mark Samuel Miller. 2006. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D.
   Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.
- Martin Mohr, Jürgen Graf, and Martin Hecker. 2015. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In Software Engineering (Workshops). Citeseer, 140–145.
- Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL* (San Antonio, Texas, USA). ACM,
   New York, NY, USA, 228–241.
- Andrew C Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. *TOSEM* 9, 4 (2000),
   410–442.
- Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In 25th USENIX Security Symposium (USENIX Security 16). 1119–1136.
   Diana Strategy Androide Difference and the second sec
  - Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. Principles of Program Analysis. Springer.
- Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical Pluggable Types
   for Java. In *ISSTA*. 201–212.
- 1703 Benjamin C Pierce. 2002. Types and Programming Languages. MIT press.
- Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type Systems for Information Flow Control: The
   Question of Granularity. ACM SIGLOG News 4, 1 (2017), 6–21.
- Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In
   2018 IEEE 31st Computer Security Foundations Symposium (CSF). IEEE, 233–246.
- Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine Grained Decentralized Information Flow Control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–74.
- Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *FormaliSE*. To appear.
- 1711 Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In CSF. IEEE, 186–199.
- 1712 Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. J-SAC 21, 1 (2003), 5–19.
- 1713Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. Journal of Computer Security 17, 51714(2009), 517–548.
- 1715

- Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W Watson. 2018. Towards Confidentiality-by-Construction. In ISoLA (LNCS, Vol. 11244). Springer, 502-515.
- Vincent Simonet. 2003. Flow Caml in a Nutshell. In APPSEM-II. 152-165.
- Alley Stoughton, Andrew Johnson, Samuel Beller, Karishma Chadha, Dennis Chen, Kenneth Foner, and Michael Zhivich. 2014. You Sank My Battleship! A Case Study in Secure Programming. In Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (Uppsala, Sweden) (PLAS'14). Association for Computing Machinery, New York, NY, USA, 2-14.
- Martin Strecker. 2003. Formal Analysis of an Information Flow Type System for MicroJava. Technische Universität München, Tech. Rep (2003).
- Qi Sun, Anindya Banerjee, and David A Naumann. 2004. Modular and Constraint-Based Information Flow Inference for an Object-Oriented Language. In SAS, Vol. LNCS. Springer, 84-99.
- Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In SAS, Vol. LNCS. Springer, 352-367.
- Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-based Deductive Verification of Software Product Lines. In GPCE. 11-20.
- Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From Fine-to Coarse-Grained Dynamic Information Flow Control and Back. Proceedings of the ACM on Programming Languages 3, POPL (2019), 1-31.
- Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. JCS 4, 2/3 (1996), 167-188.
- Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 18-35.
- Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. Acm Sigplan Notices 50, 4 (2015), 503-516.
- Lantian Zheng, Stephen Chong, Andrew C Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Aecure Distributed Systems. In 2003 Symposium on Security and Privacy, 2003. IEEE, 236-250.

1:36