LiSA: A Generic Framework for Multilanguage Static Analysis

Luca Negrini, Pietro Ferrara, Vincenzo Arceri, Agostino Cortesi

Abstract Modern software engineering revolves around distributed applications. From IoT networks to client-server infrastructures, the application code is increasingly being divided into separate sub-programs interacting with each other. As they are completely independent from each other, each such program is likely to be developed in a separate programming language, choosing the best fit for the task to at hand. From a static program analysis perspective, taking on a mixture of languages is challenging. This paper defines a generic framework where modular multilanguage static analyses can be defined through the abstract interpretation theory. The framework has been implemented in LiSA (**Li**brary for **S**tatic **A**nalysis), an open-source JAVA library that provides the complete infrastructure necessary for developing static analyzers. LiSA strives to be modular, ensuring that all components taking part in the analysis are both easy to develop and highly interchangeable. LiSA also ensures that components are parametric to all language-specific features: semantics, execution model and memory model are not directly encoded within the components themselves. A proof-of-concept instantiation is provided, demonstrating LiSA's capability to analyze multiple languages in a single analysis through the discovery of an IoT vulnerability spanning C++ and JAVA code.

Luca Negrini
Ca' Foscari University of Venice, Venezia, Italy, and Corvallis SRL, Padova, Italy e-mail: `luca.negrini@unive.it`

Pietro Ferrara
Ca' Foscari University of Venice, Venezia, Italy e-mail: `pietro.ferrara@unive.it`

Vincenzo Arceri
University of Parma, Parma, Italy e-mail: `vincnenzo.arceri@unipr.it`

Agostino Cortesi
Ca' Foscari University of Venice, Venezia, Italy e-mail: `cortesi@unive.it`

1 Introduction

Software governs most aspects of everyday life. Almost every human action, regardless of it being for work or leisure, involves at least one device that is running a program. Proving these programs correct is as important as ever, as they can collect all sorts of sensitive information (for instance, contents of medical records) or govern critical processes (like driving a car). Software architecture has dramatically evolved in the last decades. The classical client-server architecture, that was characteristic of web applications, has recently seen broader adoption with the advent of mobile applications. Moreover, the commercial drive to the *Software as a Service (SaaS)* [36], where vendors only distribute simple clients to customers while keeping all of the application logic remote, led to a huge increase of cloud computing solutions. Since clients and servers have very different purposes, the programming languages used to implement them are typically different.

Backend is also becoming less and less monolithic. Recent years have seen the rise of *microservices* infrastructures [10], where the atomic entity that was the server is split into smaller independent components that communicate with each other through APIs. Backend logic has also started being implemented through *serverless applications*, that is, code that runs in the cloud with (close to) no knowledge about the environment it runs into. Partitioning the server code into isolated entities also loosen the requirement of having those entities written in the same language, as different tasks might exploit different languages' peculiarities. One more possible segmentation of the backend comes with *blockchain oriented applications* that interact with code present on a blockchain [42]. Smart contracts are usually written with specific DSLs (e.g., SOLIDITY) dedicated to a particular blockchain in order to exploit its capabilities. Only recently a stream of blockchains adopted general purpose languages for writing smart contracts [3, 7, 32, 33].

Besides the transformation of client-server architectures, the Internet of Things (IoT) has also risen in popularity. Devices running embedded software can interact with various backends or other devices. IoT networks are becoming wildly adopted in several areas [52]: healthcare, smart homes and manufacturing are just few of the scenarios where they are applied. Once more, different programming languages can (and likely will) be involved in the realization of the system.

1.1 An illustrative example

Consider for instance the following minimal example. The code reported in Figure 3[1] has been used to prove the usefulness of static analysis for discovering IoT vulnerabilities in [24]. The code implements a system composed of a joystick and a robotic car that interact through a gateway. The JAVA fragment in Figure 1, running on the gateway, initializes the whole system and then repeatedly queries the

[1] Available at `https://github.com/amitmandalnitdgp/IOTJoyCar`.

```java
class JoyCar {
  public native int readUpDown();
  public native void runMotor(int value);
  public static void main(String[] args) {
    JoyCar rc = new JoyCar();
    //Initialization
    ...
    while(true){
      rc.runMotor(rc.readUpDown());
      //Turn based on joystick input
      ...
    }
  }
}
```

Fig. 1: Java code

```cpp
JNIEXPORT jint JNICALL Java_JoyCar_readUpDown(JNIEnv *env, jobject o){
  return readAnalog(A1);
}
long map(long val, long fl, long fh, long tl, long th){
  return (th - tl) * (val - fl) / (fh - fl) + tl;
}
void motor(int ADC){
  int value = ADC - 130;
  softPwmWrite(enablePin, map(abs(value), 0, 130, 0, 255));
}
JNIEXPORT void JNICALL Java_JoyCar_runMotor(JNIEnv *env, jobject o, jint val){
  motor(val);
}
```

Fig. 2: Embedded C++ code

Fig. 3: Excerpt of the JoyCar source code

joystick for steer and throttle. The C++ fragment in Figure 2 instead, implementing the remaining two components, interacts with the joystick's sensors and the car's motor. The two fragments communicate through the *Java Native Interface* (JNI). Here, the authors are interested in detecting the IoT Injection attack that can happen if the sensors' outputs, that could be tampered with, can *flow* into the motor's input without being sanitized, exposing the car to attacks that could damage it. Authors resort to *Taint* analysis [21, 51] for the task, but they require more than one analyzer: since the flow might span between the two codebases, analyses for Java and C++ are needed. Julia and CodeSonar were selected for the task, as they both were equipped with configurable *Taint* analysis engines able to receive a specification of sources and sinks from the user. The authors' overall analysis proceeds as follows:

1. the value returned by function `readAnalog` was marked as source of tainted information for CodeSonar;
2. the second parameter of function `softPwmWrite` was marked as a sink for tainted information for CodeSonar;
3. to detect tainted values flowing from C++ to Java code, the value returned by `Java_JoyCar_readUpDown` was marked as sink for CodeSonar;

4. to detect tainted values flowing from Java to C++ code, the first parameter of `JoyCar.runMotor` was marked as sink for Julia;
5. a first round of *Taint* analysis was run with both analyzers: Julia did not find any vulnerability (as no sources were present under Java), but CodeSonar did find a flow of tainted data going into `Java_JoyCar_readUpDown`;
6. a second round was run after marking `JoyCar.readUpDown`'s return value as source for the Java analysis, and this time Julia detected a flow of tainted data going into the first parameter of `JoyCar.runMotor`;
7. a third and final round was run after marking `Java_JoyCar_runMotor`'s third parameter as source for CodeSonar, that was now able to detect the IoT Injection vulnerability with `softPwmWrite` as sink.

Despite the successful discovery of a vulnerability that spanned multiple languages, the limits of this approach are quite evident: since tools need to exchange information at each iteration, multiple rounds of analysis are needed to reach a fixpoint over the shared information, each composed by one analysis for each tool involved. Moreover, tool communication is hard, even more if those come from different vendors as they might not agree on how information is exported and imported. Furthermore, communicating the information might not be an easy task. In this example, the authors focused on a "binary" property: a value is either tainted or not. However, with more complex structures (e.g., `Polyhedra` [17]) finding the appropriate format to exchange information between analysis might not be trivial.

1.2 Contribution and paper structure

This paper formalizes the structure of LiSA, a **Li**brary for **S**tatic **A**nalysis aimed at achieving multilanguage analysis that can be used to create static analyzers by abstract interpretation [14, 15]. Roughly, LiSA provides the full infrastructure of a static analyzer: starting from an intermediate representation in the form of extensible control flow graphs (*CFG*s), LiSA lets users define analysis components, and then takes care of orchestrating the analysis using a unique fixpoint algorithm over *CFG*s. Moreover, parsing logic is left to the user, that will define *frontend*s translating source code into *CFG*s (modeling the syntax of the input program), while also providing rewriting rules for each *CFG* node into *symbolic expression*s, an internal extensible language representing atomic semantic operations (thus modeling the semantics of each instruction of input program). We then provide a proof-of-concept multilanguage analysis using LiSA on the example reported in Section 1.1. LiSA comes as a Java library available on GitHub[2].

The remainder of this paper is structured as follows. A high-level overview is first introduced in Section 2, depicting the role of all the analysis components and how they cooperate to perform analyses. *CFG*s and *symbolic expressions* are discussed

[2] `https://github.com/lisa-analyzer/lisa`.

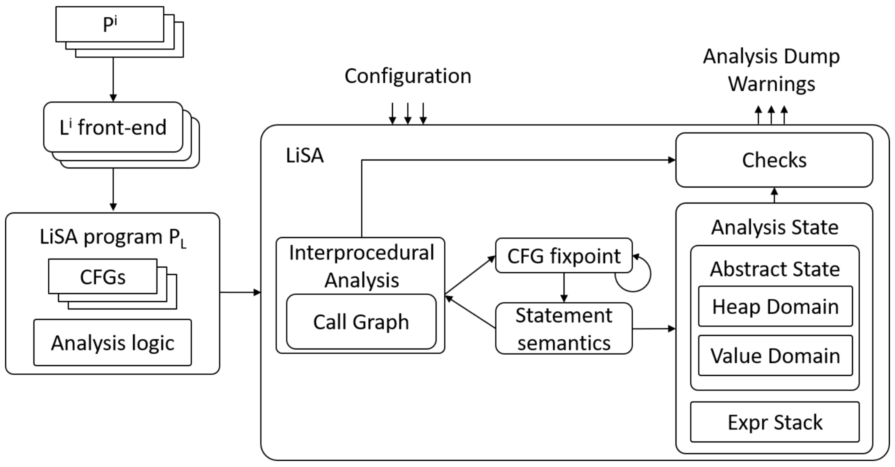

Fig. 4: LiSA's architecture

in Section 3, defining the languages that LiSA uses for syntax and semantics, respectively. Section 4 describes the modular *Analysis State* used represent pre- and post-states. The *Interprocedural Analysis* that orchestrates LiSA's fixpoint is presented in Section 5. In Section 6, we define the role of *frontend*s in compiling the source programs into LiSA's *CFG*s. We conclude the paper with a proof-of-concept multilanguage analysis on the IoT network of Section 1.1 in Section 7. An in-depth discussion of each component is available in [39].

2 LiSA's overall architecture

We begin by providing a high-level overview of the analysis pipeline, that is shown in Figure 4. The analysis starts by logically partitioning the input application P into programs P^i, each written in a single programming language L^i. L^i-to-*CFG* compilers, called *frontend*s (top-left corner of Figure 4), are invoked on such programs to obtain a uniform representation of all the code to analyze in the form of a LiSA program P^i_L. *Frontend*s are more than compilers, as they also provide logic to the analysis, such as language-specific semantics of the instructions present in *CFG*s, semantic models for library code, and language-specific algorithms for common language features (e.g., call resolution and inheritance rules). The final version P_L of the translated program to analyze is the union of all P^i_L. At this point, LiSA can be invoked on P_L with a configuration of the analysis features and the implementations of the various components that are to be executed, namely:

- the *Interprocedural Analysis*, responsible for the computation of the overall program fixpoint and for computing the results of function calls;

- the *Call Graph*, used by the *Interprocedural Analysis* to find call targets;
- the *Abstract State*, that computes the abstract values during the analysis;
- the set of *Check*s, that produce warnings based on the result of the analysis.

P_L is fed to the *Interprocedural Analysis* (left-most block within LiSA in Figure 4), that will compute a fixpoint over it. When the *Interprocedural Analysis* needs to analyze an individual *CFG*, it will invoke a unique fixpoint algorithm defined directly on *CFG*s (central portion of LiSA in Figure 4). As the language-specific semantics of instructions is embedded in *CFG* nodes, called `Statement`s, the fixpoint algorithm will use such semantics as *transfer function*. If the `Statement` performs calls as part of its semantics, it will interact back with the *Interprocedural Analysis* to determine the returned values, as how those are evaluated depends directly on how the overall fixpoint is computed. If the call's targets are unknown (for instance, if the call happens in a language with dynamic method dispatching), the *Interprocedural Analysis* can delegate targets resolution to the *Call Graph* (inner component of *Interprocedural Analysis* in Figure 4), that will use type information together with language-specific execution model to determine all possible targets. Alternatively, a non-calling `Statement`'s semantics can also rewrite the node into a sequence of *symbolic expression*s, that is, atomic instructions with precise semantics, that can be passed to the *Analysis State* for evaluation. LiSA's *Analysis State* (right-most block of LiSA in Figure 4) is composed by an *Abstract State* modeling the memory state at a given program point, together with a collection of *symbolic expression*s that are left on the stack after evaluating it. An *Abstract State* is a flexible entity whose main duty is to make the *Value Domain*, responsible for tracking values of program variables, communicate with the *Heap Domain*, that instead tracks how the dynamic memory of the program evolves at runtime. Whenever an expression is to be evaluated by an *Abstract State*, the latter first passes it to the *Heap Domain*, that will record all of its effects on the memory, such as the allocation of new regions or the access to some object's fields. Then, the *Heap Domain* will rewrite all portions of the original expression that deal with the memory. According to the implementation-specific logic, one or more instrumented variables will be used to replace such portions, modeling their resolution to memory addresses that can be treated as regular variables. After the rewriting has been performed, the resulting expression will be passed to the *Value Domain*, that will track values and properties regarding the variables appearing in it. Note that, with this architecture, each component simplifies the program for the rest of the analysis pipeline. *Interprocedural Analysis* abstracts away calls from the program to analyze, leaving the *Analysis State* and its sub-components with non-calling programs. Successively, the *Heap Domain* removes every expression that deals with dynamic memory, substituting it with synthetic variables. At this point, the *Value Domain* has to deal with programs containing only variables, constants, and operators between them.

When an overall fixpoint is reached, the computed pre- and post-states for each `Statement`, together with the *Call Graph* that has been built up, are passed to the *Check*s (top-right corner within LiSA in Figure 4) that have been provided to the analysis. These are simply visitors of the program's syntax, that can use the

```
1  class A {
2    int f, g;
3    void main(String[] args) {
4      A a = new B();
5      a.foo(10);
6    }
7    int foo(int w) {
8      return w + 2;
9    }
10 }
11
12 class B {
13   int foo(int w) {
14     this.f = w + 3;
15     this.g = this.f * 2;
16     return this.g + this.f;
17   }
18 }
```

Fig. 5: Running example for LiSA's architecture overview

information computed by the analysis to issue warnings for the user. Since these are a standard component of static analyzers, they will be omitted by this work.
Example. Consider the example Java code from Figure 5. To analyze it, a Java *frontend* will first parse the code and produce three different *CFG*s, one for each method. Supposing that the *Interprocedural Analysis* applies `context-sensitive` [46] reasoning, the analysis could follow call-chains starting from the `main` *CFG*, analyzing *CFG*s are they are invoked. Thus, the first fixpoint algorithm to be invoked would be the one of `main`. Here, when the call to `foo(10)` at line 5 is encountered, an entry state for the targets of the call would be set up by assigning 10 to `w`. How the call would be resolved to its targets depends once again on the configuration. Supposing that the *Call Graph* relies on the runtime types inferred for the receiver [48], the call will be resolved to `B.foo` that can be analyzed (that is, whose fixpoint can be executed) using the prepared state. The code of `B.foo` deals with heap structures. The assignment at line 15 could be rewritten as `l_0 = l_1 * 2` if the *Heap Domain* is precise enough to distinguish between different fields of the same object, or as `l_0 = l_0 * 2` if it is not. Nonetheless, the resulting expression will not contain any reference to memory structures, and can be then processed by the *Value Domain* (e.g., intervals [14]) agnostically w.r.t. if and how a rewriting happened.

3 The internal language

Before detailing the separate components of LiSA, we introduce and discuss (i) the *CFG* structure that LiSA uses for representing programs, and (ii) the *symbolic expression* language used as intermediate representation for analysis. As programming languages come with wildly different syntax, it is important to find a common ground to model their semantics so that analyses are not required to handle constructs from all languages. In fact, this is a common practice among static analyzers, even ones targeting a single language: moving to a uniform and more convenient intermediate representation (IR) that is usually enriched with additional information (e.g., dynamic typing) can make writing analyses much easier. Different syntactic constructs with the same (or similar) meaning can be represented by the analyzer as a unique IR construct, and complex ones can be decomposed as a sequence of them. Analyses can then attribute semantic meanings to such IR constructs with no knowledge of the original syntactic ones they represent.

Rewriting towards the IR can typically be achieved at parse time, after ingesting the target application, or at analysis time, before passing the code to the abstract domains. LiSA implements hybrid rewriting: first, source code is compiled to *control flow graphs* (*CFG*s) by *frontend*s, then each *CFG* node is rewritten into one or more *symbolic expression*s during the analysis. *CFG*s thus embed syntactic structures and language-specific constructs within them (i.e., `+` is a syntactic construct that might represent numeric addition, string concatenation, ...), while each *symbolic expression* has a unique semantic meaning.

LiSA's programs are thus composed of *CFG*s, that can be logically grouped in `CompilationUnit`s, a generalization of the concept of *classes* in object-oriented software. As both the structure and meaning of `CompilationUnit`s mirrors the one of classes (with some additional parametrization for language-specific features like multiple inheritance), and does not directly influence the infrastructure of LiSA, their definition is omitted in this work. Thus, we will refer to LiSA programs as a collection of *CFG*s.

3.1 Control flow graphs

Control flow graphs [1] (CFG) are directed graphs that express the control flow of functions. In a CFG, nodes contain the instructions of the functions, and edges express how the execution *flows* from one node to another. This means that all the syntactic constructs that form loops, branches and arbitrary jumps are directly encoded in the CFG structure, simplifying the code to analyze. LiSA's *CFG*s are extensible: `Statement` and `Edge` are the base definitions of what nodes and edges are, respectively, while concrete instances are defined in *frontend*s.

A `Statement` (base class for *CFG* nodes) represents an instruction appearing in a function, and thus corresponds to a syntactic construct that does not modify the control flow (that is, it is not a loop, a branch or an arbitrary jump). When the evaluation of a `Statement` leaves a value on the operand stack, it is called an `Expression` whose type is the one of the generated value. Examples of `Statement` are `return` and `assert`, while an `Expression` can be a reference to a local variable by its name, an assignment, or a sum. The `Call` expression, together with its descendants, plays a central role in LiSA and will be further discussed in Section 5. Note that `Statement`s do not have a predefined semantics: in fact, the class defines a `semantics` method where implementers can define language-specific reasoning and interact with `entryState` (instance of *Analysis State* representing the pre-state) and `interproc` (instance of *Interprocedural Analysis* offering interprocedural reasoning) to compute the post-state for the `Statement`.

The `Edge` class is the base class for *CFG* edges. The `traverse` method defined by this class expresses how the post-state of its source node, in the form of an *Analysis State* instance, is transformed when the edge is traversed. LiSA comes with three default implementations for edges: `SequentialEdge`, `TrueEdge` and `FalseEdge`. `SequentialEdge` represents an unconditional flow of execution from source to edge, with no modification to the initial *Analysis State* (i.e., its `traverse` implementation returns its parameter unaltered). `TrueEdge` instead models a flow of execution conditional to the evaluation of the expression at its source: the execution proceeds by reaching the edge's destination only if it evaluates to `true`. Conversely, `FalseEdge` implements a conditional flow of execution that reaches the edge's destination only if the expression at its source evaluates to `false`.

Finally, LiSA offers *native CFG*s, that is, CFGs with a single `Statement` and no `Edge`s, as a mean to model individual library functions. Whenever a call to one of these CFGs is found, the call's result can be evaluated by rewriting it into the only statement contained in the *native CFG*, and then executing its `semantics` method. Modeling complex or frequently used library functions through *native CFG*s can drastically reduce the complexity of the analysis, as less code needs to be analyzed, while still providing all the necessary information about the modeled functions.

3.2 Symbolic expressions

A static analyzer's main duty is to compute program properties by taking into account the semantic meaning its instructions. In this context, an extensible set of syntactic constructs such as the one provided by LiSA through *CFG*s comes with an intrinsic problem: instructions (i.e., `Statement`s) do not have a well-defined semantics, as that is parametric to the source language. To recover well-definedness, LiSA adopts a two-phases rewriting: not only is the source program compiled to *CFG*s, but each of their `Statement` gets rewritten into *symbolic expression*s during the analysis.

The `SymbolicExpression` class is the base type for the expressions that LiSA's analysis components understand and analyze. Note that there is a clear distinction between expressions dealing with values of variables (i.e., `ValueExpressions`) and ones dealing with memory structures (i.e., `HeapExpressions`). This is a direct consequence of the architecture, introduced in Section 2 and discussed in Section 4, that separates domains dealing with the two worlds, decoupling their implementations. `ValueExpression`s model what can be handled entirely by the *Value Domain*: constants, variables, and operators between them. On the other side, `HeapExpression`s represent operations that change or navigate the structure of the dynamic memory of the program. As for *CFG*s, `Statement`s and `Edge`s, *symbolic expression*s are also extensible. Note that no *symbolic expression* is defined for calls, as those are abstracted away by the *Interprocedural Analysis* (Section 5).

To better explain how the second rewriting is carried out, let us consider the expression `new B()` at line 5 of Figure 5. In Java, object instantiation consists of four operations: (i) allocation of a memory region, (ii) creation of a pointer to the region, (iii) invocation of the desired constructor using the fresh pointer as receiver, and (iv) storage of the pointer on the operand stack. Such behavior could be mimicked by a `Statement` instance with the following (simplified) `semantics` function:

```
1  AnalysisState semantics(AnalysisState entry, InterproceduralAnalysis iproc) {
2    // create a synthetic receiver
3    VariableRef rec = new VariableRef("$receiver");
4    AnalysisState recState = rec.semantics(entryState, interproc);
5    // assign the fresh memory region to the receiver
6    HeapReference ref = new HeapReference(new HeapAllocation());
7    AnalysisState callState = entryState.bottom();
8    for (SymbolicExpression v : recState.getComputedExpressions())
9      callState = callState.lub(callstate.assign(v, ref));
10   // call the constructor
11   String name = createdType.toString();
12   Expression[] params = ArrayUtils.insert(0, expressions, rec);
13   UnresolvedCall call = new UnresolvedCall(name, name, params);
14   AnalysisState sem = call.semantics(callState, interproc);
15   // leave a reference on the stack
16   return sem.smallStepSemantics(ref);
17 }
```

While the methods exploited in this snippet are the subject of the following sections, we can nonetheless capture the intuition behind them. First, a `VariableRef` (that is an instance of `Statement` and thus modeling the syntactic reference to a program's variable by its name) is created at line 3, mimicking the creation of the constructor call's receiver. Then its semantics is computed at line 4 starting from the pre-state `entry`, obtaining a new instance of *Analysis State* (Section 4) that will contain the `Variable` (an instance of `SymbolicExpression`) corresponding to it. The following five lines are responsible for making such variable point to a newly allocated memory region: line 9 assigns a pointer (line 6) to a region that is being allocated in-place to the `Variable` corresponding to the receiver (line 8). Next, the call to the constructor is performed at line 14 by (i) extracting the name of the type created by the expression (line 11, where `createdType` is a field containing the `Type` being instantiated) as both the class and method name, and (ii) adding the instrumented receiver to the original parameters of the constructor call (line 12, where `expressions` is a field containing the original parameters and `ArrayUtils.insert` is a method that clones an array and adds a new element to it). The semantics of the `UnresolvedCall` of line 14 will defer the resolution and evaluation to the *Interprocedural Analysis*. The post-state of the call is then used at line 16 to reprocess the reference to the memory region through `smallStepSemantics`, returning the result as the final post-state of the whole instruction.

4 The Analysis State

The state of LiSA's analyses is modularly built bottom-up, ensuring that each component does not have visibility of its parents and siblings. Modularity entails that no additional knowledge is needed to implement a component other than what is strictly required by it. As presented in [26], this is crucial for picking up LiSA quickly. Each instance of *Analysis State* represent both elements of an ordered structure (thus exposing methods for partial ordering, least upper bound, . . .), and abstract transformers able to produce new elements (offering methods such as `assign` and `smallStepSemantics` that evaluate the semantics of a *symbolic expression*).

We briefly discuss each internal component of the *Analysis State*, introducing them bottom-up. A full discussion of each can be found in [39]. LiSA's *Value Domain* is the analysis component responsible for tracking properties about program variables, and it is the *last* component taking action to compute a *symbolic expression*'s semantics. Examples *Value Domain* implementations are `Interval` [14], `Polyhedra` [17], and `Tarsis` [40], or their combinations (e.g., products [11] and smashed sums [4]). LiSA provides a *Value Domain* implementation named `ValueEnvironment` defining the point-wise logic for Cartesian (i.e., non-relational) abstractions. Domains such as `Interval` can thus be implemented by (i) providing lattice operations for single intervals, and (ii) defining the logic for expression evaluation. LiSA then takes care of wrapping the domain inside a `ValueEnvironment`, providing a unique functional lifting to all Cartesian abstractions. An example implementation is presented in [26].

LiSA's *Heap Domain* is the analysis component responsible for tracking how the dynamic memory of the program evolves during its execution. As the sole component having full knowledge on how expressions are resolved to memory locations, the *Heap Domain* operates before the *Value Domain* as it must simplify memory-dealing expressions that the latter cannot handle. Examples *Heap Domain* implementations are Andersen's `Pointer Analysis` [2] and `Shape` analysis [45]. Moreover, as for *Value Domain*s, combinations of *Heap Domain*s are still *Heap Domain*s.

LiSA's *Abstract State* wraps both *Value* and *Heap Domain*s, coordinating their communication. It is designed after the framework presented in [22], where the two communicate by means of expression rewriting and variable renaming. Roughly, the semantics of such framework lets the two domains compute properties independently whenever an expression only deals with values or memory. When one instead requires knowledge about both worlds, the expression is first evaluated by the *Heap Domain* that tracks its effects on the memory. Then, abstract locations called *heap identifiers* are used to replace memory-dealing sub-expressions, and the *Value Domain* can process this rewritten expression to track properties about those identifiers. Furthermore, as the semantics of the *Heap Domain* might materialize or merge heap identifiers, a *substitution* can be applied to the *Value Domain* when necessary, before computing its semantics. A substitution can be described as a sequence of multi-variable replacements between heap identifiers. The concrete implementation of the communication between the *Heap Domain* and *Value Domain* is provided by the `SimpleAbstractState` class. *Abstract State* is left modular and extensible as further layers of abstraction can be applied on the entire state. For instance, `Trace Partitioning` [43] must be applied on the state *as a whole*, and can be defined as an *Abstract State* implementing a function from execution traces to *Abstract State*s.

The *Analysis State* is the outer-most component, and it is thus the one explicitly visible to the rest of the analysis. A direct implication of this is that other compo-

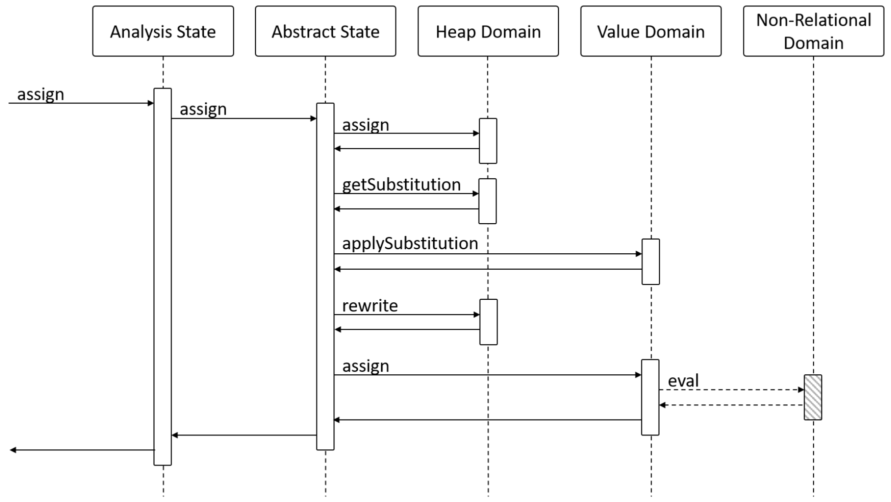

Fig. 6: Sequence diagram *Analysis State*'s `assign`

nents are agnostic w.r.t. how LiSA abstracts memory structures and values of the program. Its main duty is to wrap the *Abstract State* together with additional semantic information over which the analysis must reach a fixpoint. Here, we identify as *mandatory* information only the set of *symbolic expression*s that are left on the stack after evaluating an arbitrary expression, but more can be added.

To grasp the intuition of how the *Analysis State* operates, consider the sequence diagram of Figure 6, depicting how the `assign` method behaves. Note that the pattern shown here is also valid for the other semantic operations, as they all follow the overall communication scheme defined in [22]. When the `assign` method of the *Analysis State* is invoked, the call is immediately forwarded to the *Abstract State*. The latter will first compute the effects of the assignment on the dynamic memory through the *Heap Domain*'s own `assign` method. Then, since such operation might have caused materialization or merge of heap identifiers, the *Abstract State* retrieves a *substitution* from the *Heap Domain*, and uses it to update the *Value Domain*. Then, *Heap Domain*'s `rewrite` replaces portions of the assigned expression dealing with dynamic memory with heap identifiers, rendering the right-hand of the assignment *memory-free*. The updated *Value Domain* instance is then used to evaluate the effects of the assignment on program variables, using *Value Domain*'s `assign` method on the rewritten expressions. The new *Heap* and *Value Domain* instances are then wrapped into a fresh *Abstract State* object, that is returned to the caller as part of the final *Analysis State* that is built as result of the original call. Note that, as *Abstract State*, *Heap Domain* and *Value Domain* are defined modularly, each computational step might hide additional complexity (for instance, the *Value Domain* could be a Cartesian product of several instances, whose `assign` methods are recursively invoked). Moreover, the *Value Domain* can optionally rely on an inner *Non-Relational Domain* instance to compute the semantics of an expression, exploiting its `eval` method.

5 Interprocedural Analysis

LiSA's *Interprocedural Analysis* is responsible for computing both a program-wide fixpoint and the result of calls, as the two features are strictly related. In fact, the computation of the overall program fixpoint is a direct consequence of call evaluation. If a pre-computed result is to be returned when a call is encountered, call-chains should be analyzed bottom-up starting from the target of the last call, thus ensuring that results are already available when needed. Instead, if results are to be freshly generated, call-chains should be analyzed top-down, starting from the *CFG* containing the first call. LiSA's semantic analysis begins by querying *Interprocedural Analysis* for the program's fixpoint, that will implement a strategy for traversing call-chains and reaching a fixpoint on each of its *CFG*s. Individual *CFG* fixpoints are evaluated using the classical worklist based fixpoint over graphs, uniquely implemented in LiSA. Such algorithm exploits the language-specific `semantics` functions defined by `Statement`s. As discussed in Section 3.2, non-calling `Statement`s will rewrite themselves as a sequence of *symbolic expression*s. When a call must be evaluated, the rewriting process is not enough.

Four concrete `Call` instances are included in LiSA: `NativeCall`, `CFGCall`, `OpenCall`, and `UnresolvedCall`. `NativeCall`s only target *native CFG*s: the semantics of this call exploits the latter's rewriting functionality to transform them into a `Statement` instance, and then delegates the computation to its semantics. `CFGCall`s instead invoke *CFG*s, and thus have to access their targets' fixpoint results through the *Interprocedural Analysis*. Instead, `OpenCall`s are evaluated as an abstraction of an unknown result (for instance, one could conservatively assume that open calls can natively manipulate memory, thus always return $\top$ as post-state). Lastly, `UnresolvedCall`s only carry signature information, and are not yet bound to their targets. Target resolution is performed by the *Call Graph*, but a further degree of modularity is added by having the call interact with *Interprocedural Analysis* instead, as some implementations (i.e., intraprocedural ones) might return fixed over-approximations when calls are evaluated, bypassing the *Call Graph* invocation that otherwise happens here. Regardless, every `UnresolvedCall` is converted to one of the other `Call` instances[3], and its semantics can be normally applied.

Interprocedural Analysis implementations can be `context-sensitive` [31,46], following call-chains top-down evaluating each *CFG* fixpoint as they are called, or `modular` [16] (also called `summary-based`), where call-chains are analyzed bottom-up accessing pre-computed results.

The *Call Graph* is tasked with resolving `UnresolvedCall`s to their targets. As such calls only come with signature information (i.e., the name of the target *CFG*)

[3] `UnresolvedCall`s might resolve to both *CFG*s and *native CFG*s: here, LiSA instantiates a `MultiCall`, whose semantics yields the lub of the internal `CFGCall` and `NativeCall`s.

and their parameters, the whole program needs to be searched for possible targets. The search is a complex operation that relies on several features of programming languages, and it can be logically split into two phases: scanning for possible targets along the program and matching the actual parameters to each candidate's signature.

Candidate scanning depends on the call type. If the call is known to have a receiver (that is, if it is an *instance* call), only the receiver's type hierarchy is to be searched for targets. Hierarchy traversal is language-specific, as it is influenced by how the language enables inheritance (e.g., it might be single or multiple, or it might provide explicit and implicit interface implementations). Instead, choosing the starting point for hierarchy traversal is a feature depending on the implementation of *Call Graph*: for instance, a graph implementing `Class Hierarchy analysis` [18] would consider all possible subtypes of the receiver's static type, while one implementing `Rapid Type analysis` [5] would restrict such set to the subtypes instantiated by the program. Regardless, the hierarchy of candidate types must then be searched for *CFG*s with a matching name. If the call instead does not have a receiver (i.e., if it is *static*), the whole program needs to be searched for *CFG*s with a matching name.

Once candidates have been selected, the actual parameters of the call must be matched with the formal ones defined by each target. Once more, this feature is language-specific: capabilities like optional parameters and named ones, as well as how types of each parameter are evaluated complicate the matching process to a point where no unique algorithm can be applied. To make resolution parametric, LiSA delegates language-specific call resolution features to each `UnresolvedCall` instance: when created, users need to specify both a strategy for traversing a type hierarchy given a starting point, and a strategy for matching a list of `Expression`s to an arbitrary signature. Note that, once a call has been resolved, an entry state for the targets has to be prepared by assigning actual parameters to formal ones. As this process also follows the parameter matching, the *Interprocedural Analysis* will defer this preparation to the same algorithm.

6 Frontends

*Frontend*s are tasked with performing the first rewriting phase, translating a (possibly partial) source program into one made of *CFG*s that can be analyzed by LiSA. As mentioned at the beginning of Section 3, a component performing a translation is included in most static analyzer, as moving to a more convenient representation makes writing analyses simpler. LiSA's *frontends*, however, are more than just raw compilers: as the sole component with deep knowledge about the language they target, they must define `Statement`s with their semantics, types, and language-specific algorithms that implement the execution model of the language.

Even if they might seem less relevant to the whole analysis process, writing and maintaining a complete *frontend* for a language is no easy endeavor. In fact, mature and widespread programming languages have a very complex semantics to model, with features that might be ambiguous or not formally defined[4]. Moreover, each language has its own evolution, leading to different versions needing support. This not only translates to a higher number of instructions to model, but also to a variety of runtime environments (containing different libraries and software frameworks) whose semantics has to be taken into account for precise static analysis.

Writing a *frontend* usually begins with the code parsing logic. Whenever it is not possible to use official tools (e.g., by plugging into the compilation process), parser generators such as ANTLR[5] can be used to create custom abstract syntax tree visitors. `Statement` instances for each instruction must be included as part of the *frontend* (potentially using common implementations provided by LiSA), each one providing its own semantics and bringing language-specific algorithm that are exploited during the analysis. Type inference is optional, as the one ran by LiSA during the analysis can be exploited inside semantics functions. This means that constructs such as +, that in most languages have different semantics depending on the type of its operands, can be modeled by a single `Statement` instance. Modeling runtimes and libraries is achieved by means of *native CFG*s or `SARL` [25].

7 Multilanguage analysis

We now instantiate LiSA and its components to showcase how multilanguage analyses can be easily defined. We demonstrate the effectiveness of our approach on the JoyCar IoT system of Figure 1.1. Code snippets reported in this section are available on GitHub[6], where the full implementation of this analysis is published.

As the codebase is composed of two languages, a *frontend* for each has to be built. These have been developed using ANTLR for parser generation, and mostly exploit `Statements` and `Edges` provided out-of-the-box from LiSA. The key aspect w.r.t. multilanguage analysis is the handling of constructs that enable inter-language communication, offered here by the *Java Native Interface* (JNI). At runtime, the Java VM tries to resolve calls to native methods using the name-mangling scheme reported in the JNI specification[7]. We thus proceeded by providing an implementation for native methods found in Java code using the following (simplified) snippet:

```
void parseAsNative(CFG cfg, String className, String name,
    Parameter[] formals, Type returnType) {
  String mangled = nameMangling(className, name, formals);
  Expression[] args = buildArguments(formals);
  UnresolvedCall call = new UnresolvedCall(mangled, args);
  if (!returnType.isVoidType())
  cfg.addNode(new Return(call));
```

[4] For instance, Python does not have a formal specification of its semantics, while C admits syntactic constructs whose behavior is undefined.

[5] `https://www.antlr.org/.`, with several well-tested grammars available at `https://github.com/antlr/grammars-v4`

[6] `https://github.com/lisa-analyzer/lisa-joycar-example`.

[7] `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html`, paragraph “Resolving Native Method Names”.

```
  else {
    Ret ret = new Ret();
    cfg.addNode(call);
    cfg.addNode(ret);
    cfg.addEdge(new SequentialEdge(call, ret));
  }
}
```

The code above bridges the two codebases by creating an `UnresolvedCall`, where (i) the target's name is built with the mangling scheme from the specification, (ii) the arguments for the call correspond to the ones passed to the native method preceded by the pointer to an instance of `JNIEnv` (an object required by JNI to hold pointers to native functions), and (iii) the value returned by the call is also returned by the native method, if any. With this setup, not only can the C++ code be parsed regularly, but the analysis components are also agnostic to the presence of JNI, as the call will to the native method is treated exactly as any other call. Note that, while this specific example did not require it, the generated call can be preceded by arbitrary instrumentations (e.g., the state conversion typical of *boundary functions*).

The next step is to select the analysis components. We mostly rely on analyses natively provided by LiSA:

- the *Interprocedural Analysis* is set to a `context-sensitive` implementation that follows call-chains top-down, thus starting from the `main` method and traversing them until a recursion is encountered (and thus enabling LiSA to follow *every* call in our target application);
- the *Call Graph* uses inferred runtime types of variables and expressions;
- the *Abstract State* used is `SimpleAbstractState`;
- as the program properties do not rely on dynamic memory, we use a fast but imprecise *Heap Domain* called `MonolithicHeap`, that abstracts each memory location to a unique synthetic one;

For *Value Domain*, we implemented a `Taint` analysis [21,51] as a *Non-Relational Domain*, whose simplified source code is reported in Figure 7. The domain is based on the poset $\langle\{\bot, clean, tainted\}, \{(\bot, clean), (clean, tainted)\}\rangle$, that forms a finite (and thus complete) lattice using trivial $\sqcup$ and $\sqcap$ operators that, given a pair of elements, return the greater and smaller of the two, respectively. Implementation-wise, the superclass `BaseNonRelationalValueDomain` handles base cases of lattice operations, that is, when one of the operands involved is either $\top$ ($tainted$) or $\bot$, or when the two operands are the same element. Hence, no additional logic needs to be implemented for $\sqcup$, ∇ and $\sqsubseteq$. Recursive expression evaluation is also provided out-of-the-box by `BaseNonRelationalValueDomain`, and the concrete implementation only has to provide evaluation of individual expressions. Specifically, we consider all constants as $clean$, while evaluation of unary, binary and ternary expressions is carried out by computing the $\sqcup$ of their arguments. Tainted values are generated only when evaluating variables, relying on their annotations: as LiSA assigns the result of `Call`s to temporary variables, transferring all annotations from the call's targets, this enables uniform identification of both sources (i.e., annotated with `TAINT_ANNOT`) and sanitizers (i.e., annotated with `CLEAN_ANNOT`). Variables are thus considered

```java
public class Taint extends BaseNonRelationalValueDomain<Taint> {
  static final Annotation TAINT_ANNOT = new Annotation("lisa.taint.Tainted");
  static final Annotation CLEAN_ANNOT = new Annotation("lisa.taint.Clean");

  static final Taint TAINTED = new Taint(true);
  static final Taint CLEAN = new Taint(false);
  static final Taint BOTTOM = new Taint(null);

  final Boolean taint;
  Taint(Boolean taint) {
    this.taint = taint;
  }

  Taint evalIdentifier(Identifier id, ValueEnvironment<Taint> env) {
    Annotations annots = id.getAnnotations();
    if (annots.contains(TAINT_ANNOT))
    return TAINTED;
    if (annots.contains(CLEAN_ANNOT))
    return CLEAN;
    return env.getState(id);
  }

  Taint evalConstant() {
    return CLEAN;
  }

  Taint evalUnaryExpression(UnaryOperator operator, Taint arg) {
    return arg;
  }

  Taint evalBinaryExpression(BinaryOperator operator,
  Taint left, Taint right) {
    return left.lub(right);
  }

  Taint evalTernaryExpression(TernaryOperator operator,
  Taint left, Taint middle, Taint right) {
    return left.lub(middle).lub(right);
  }
}
```

Fig. 7: A simple `Taint Analysis` implementation

always *tainted* or always *clean* relying on such annotations, defaulting otherwise to the abstraction stored inside the environment.

To exploit our analysis' results, we defined a *Check* instance that iterates over the application under analysis to scan for method parameters that are annotated with `@lisa.taint.Sink`, a third kind of annotation that identifies places where tainted information should not flow. When one such parameter is found, the check inspects all call sites where the corresponding method is invoked, and checks if the post-state of the `Expression` passed for the annotated parameter contains a tainted expression on the stack, according to our `Taint` domain. If it is, a warning is issued. We then proceed by annotating as source (i.e., with `@lisa.taint.Tainted`) the value returned by `readAnalog`, and as sink (i.e., with `@lisa.taint.Sink`) the second parameter of `softPwmWrite`. The analysis can then be executed, obtaining the following warning on the `softPwmWrite` call:

The value passed for the 2nd parameter of this call is tainted, and it reaches the sink at parameter 'value' of softPwmWrite [at JoyCar.cpp:124:55]

thus showcasing that cross-language vulnerabilities can be discovered in a single analysis run. Also note that, with the same setup, domains computing complex structures (e.g., automata) can still operate cross-language without incurring in expensive serializations and deserializations needed to communicate information across analyzers.

8 Conclusion

In this chapter, we thoroughly described LiSA, a modular framework for multi-language static analysis with an open-source JAVA implementation. LiSA operates by analyzing an extensible language of *CFG*s, whose node contain user-defined language-specific semantics that translate them into *symbolic expressions*. These are atomic constructs with precise semantics, that abstract domains can analyze. LiSA's infrastructure modularly decomposes semantics evaluation into separate tasks, each carried out by a different analysis component. Each such component performs agnostically w.r.t. the concrete implementations of other ones, and is responsible for abstracting specific program features. The *Interprocedural Analysis*, in cooperation with the *Call Graph*, abstracts calls and call-chains, leaving the rest of the analysis with call-free programs. Then, the modular *Analysis State* orchestrates memory and value abstraction. The former is performed by the *Heap Domain*, that abstracts all heap operations by rewriting them with synthetic variables representing heap locations they resolve to, leaving the *Value Domain* with call- and memory-free programs. Individual library functions can be modeled as *native CFG*s, that is, as special CFGs with a unique node that expresses the semantics of the whole function. We also reported our teaching experience with LiSA, that emphasizes how the modular structure enables experimenting with LiSA achievable by Master level students. Furthermore, we demonstrated LiSA's capability of analyzing software written in multiple programming languages, identifying a vulnerability that spanned JAVA and C++ in a proof-of-concept case study.

8.1 Future directions

As a multilanguage analyzer, one of our objectives is to target as many programming languages as possible. This not only means having *fronted*s for each of them, but also ensuring that the internal LiSA program model is flexible and parametric enough to represent syntactic structures, semantics, execution model, inheritance, and all of their other peculiarities. Currently, *frontend*s for GO, PYTHON, JAVA, JAVA BYTECODE, RUST, and MICHELSON BYTECODE are in development, but this is undoubtedly a long-term effort. One additional vision for LiSA's future is to not only provide users with an easy-to-use tool where new analysis can be implemented quickly and tested on several languages, but where they can also easily compare different implementations with their own. As such, we plan on extending LiSA to ship with several well-known component implementations, from numerical and string domains (e.g., `Octagons` [37] and `Bricks` [12]), to property domains (e.g., `Information Flow` [19, 44]), to interfaces with widely accepted static analysis libraries (e.g., Apron [29] and PPL [6]), also widening the analysis spectrum to backward analyses.

8.2 Related work

As the field grew and matured over decades, a vast literature about static analysis and abstract interpretation is available (most notable results are referenced in [13]), reporting a wide spectrum of techniques and their applications to prove software correct. Here, we focus on work strictly related to multilanguage analysis.

The initial focus on multilanguage analysis targeted combinations of similar languages. Julia [47] analyzes Java bytecode, and has been extended to also analyze CIL bytecode resulting from the compilation of C# code by means of a semantic translation into Java bytecode [23]. Infer [20] analyzes Java, C, C++, and Objective-C programs by statically translating them into a proprietary intermediate representation, called SIL, composed of only four instructions. However, these approaches are intrinsically limited by the expressiveness of the intermediate representation (Java bytecode and SIL in case of Julia and Infer): since the set of constructs in those representations is predefined, they might not be enough to represent features of some languages. For instance, Java bytecode cannot express pointer arithmetic, while SIL is not suited for dynamic typing.

Another stream of work instead considers a scenario where one central portion of the application, written in a single programming language, interacts with *native code*, that in this context can be considered as a collection of functions written in different programming languages. [53] performs a summary-based analysis of Android applications as a whole (that is, Java code and JNI-exposed native code), while [49] compiles C code into an extended Java bytecode that can be analyzed by existing Java analyzers. [27,28] perform type inference on so-called *foreign function interfaces*, that is, inter-language communication frameworks like JNI, discovering type errors otherwise visible only at runtime. [35] instead detects mishandling of exceptions and errors raised in native code and then propagated back to Java. The work presented in [34] computes semantics summaries from *guest* programs, to be used during the analysis of a *host* program. All of these approaches share the same underlying idea: to compute summaries among a family of "secondary" programs and use them to analyze the main one (here, programs are defined as coherent modules written in the same language). While modular summary-based analyses are powerful and scalable, not all properties can be proven bottom-up, and often require precise context-sensitive analyses. Moreover, not all programs can be described as a single processing entity exploiting auxiliary codebases: for instance, mobile apps contain logic on both the app and the backend, with back-and-forth communication between the two.

More recently instead, solutions for multilanguage analyses have been proposed. [8, 9] provide an algebraic framework for multilanguage abstract interpretations by combining existing language-specific ones through *boundary functions* that perform state conversion when switching context between languages. [50] provides LARA [41] source-to-source compilation to transform JAVA, C, C++, and JAVASCRIPT programs towards a common syntax, over which static analyses can be run. Authors however focus on source-code metrics (that is, syntactic properties), with no reasoning on runtime behaviors (that is, semantic properties).

The major alternative to the approach described in this paper is undoubtedly Mopsa [30] (**M**odular **O**pen **P**latform for **S**tatic **A**nalysis), a static analyzer based on the abstract interpretation theory written in OCAML. Mopsa is designed to compute fixpoints by induction on a program's syntax. A program is an extensible abstract syntax tree (AST) that initially contains the original source code, but that can be syntactically and semantically rewritten during the analysis. Abstract domains share a common interface, and are thus easy to compose and extend. Moreover, the domains are responsible for dynamically rewriting fragments of the AST exploiting semantic information, avoiding static translation towards an internal language. Depending on both the target programming languages and the properties of interest, Mopsa's analyses need to be configured by composing a chain of abstract domains that will dynamically rewrite portions of the AST until a common syntax is reached, over which the remaining domains can operate independently from the source language. Mopsa has been successfully used to analyze C and PYTHON programs [30], showcasing its ability to target completely different languages, including dynamic ones. Moreover, analyses on a combination of the two have been performed [38].

Acknowledgments

Work partially supported by SPIN-2021 projects "Ressa-Rob" and "Static Analysis for Data Scientists”, and Fondi di primo insediamento 2019 project "Static Taint Analysis for IoT Software" funded by Ca’Foscari University, and by iNEST-Interconnected NordEst Innovation Ecosystem, funded by PNRR (Mission 4.2, Investment 1.5), NextGeneration EU - Project ID: ECS 00000043."

References

1. Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization, p. 1–19. Association for Computing Machinery, New York, NY, USA (1970). DOI 10.1145/800028.808479. URL `https://doi.org/10.1145/800028.808479`
2. Andersen, L.O.: Program analysis and specialization for the c programming language. Ph.D. thesis, DIKU, University of Copenhagen (1994). URL `https://citeseerx.ist.psu.edu/`

viewdoc/download?doi=10.1.1.109.6502&rep=rep1&type=pdf

3. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A.D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S.W., Yellick, J.: Hyperledger fabric: A distributed operating system for permissioned blockchains. In: R. Oliveira, P. Felber, Y.C. Hu (eds.) Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, pp. 30:1–30:15. ACM (2018). DOI 10.1145/3190508.3190538
4. Arceri, V., Maffeis, S.: Abstract domains for type juggling. Electron. Notes Theor. Comput. Sci. **331**, 41–55 (2017). DOI 10.1016/j.entcs.2017.02.003
5. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '96, p. 324–341. Association for Computing Machinery, New York, NY, USA (1996). DOI 10.1145/236337.236371. URL https://doi.org/10.1145/236337.236371
6. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1), 3–21 (2008). DOI 10.1016/j.scico.2007.08.001. URL https://www.sciencedirect.com/science/article/pii/S0167642308000415. Special Issue on Second issue of experimental software and toolkits (EST)
7. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis, University of Guelph (2016). URL https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769
8. Buro, S., Crole, R.L., Mastroeni, I.: On multi-language abstraction. In: D. Pichardie, M. Sighireanu (eds.) Static Analysis, pp. 310–332. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-65474-0_14
9. Buro, S., Mastroeni, I.: On the multi-language construction. In: L. Caires (ed.) Programming Languages and Systems, pp. 293–321. Springer International Publishing, Cham (2019). DOI 10.1007/978-3-030-17184-1_11
10. Chen, L.: Microservices: Architecting for continuous delivery and devops. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 39–397 (2018). DOI 10.1109/ICSA.2018.00013
11. Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. Electronic Proceedings in Theoretical Computer Science **129**, 325–336 (2013). DOI 10.4204/eptcs.129.19. URL https://doi.org/10.4204%2Feptcs.129.19
12. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Software: Practice and Experience **45**(2), 245–287 (2015). DOI 10.1002/spe.2218. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2218
13. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: R.M. Graham, M.A. Harrison, R. Sethi (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977). DOI 10.1145/512950.512973
15. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: A.V. Aho, S.N. Zilles, B.K. Rosen (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, pp. 269–282. ACM Press (1979). DOI 10.1145/567752.567778
16. Cousot, P., Cousot, R.: Modular static program analysis. In: R.N. Horspool (ed.) Compiler Construction, pp. 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). DOI 10.1007/3-540-45937-5_13
17. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: A.V. Aho, S.N. Zilles, T.G. Szymanski (eds.) Conference Record of the Fifth

Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pp. 84–96. ACM Press (1978). DOI 10.1145/512760.512770. URL `https://doi.org/10.1145/512760.512770`

18. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: M. Tokoro, R. Pareschi (eds.) ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995, pp. 77–101. Springer Berlin Heidelberg, Berlin, Heidelberg (1995). DOI 10.1007/3-540-49538-X_5
19. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976). DOI 10.1145/360051.360056. URL `https://doi.org/10.1145/360051.360056`
20. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Communications of the ACM **62**(8), 62–70 (2019). DOI 10.1145/3338112
21. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in java. In: M. Davis, A. Fehnker, A. McIver, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, pp. 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). DOI 978-3-662-48899-7_10
22. Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. Theoretical Computer Science **631**, 43–72 (2016). DOI 10.1016/j.tcs.2016.04.001. URL `https://www.sciencedirect.com/science/article/pii/S0304397516300299`
23. Ferrara, P., Cortesi, A., Spoto, F.: Cil to java-bytecode translation for static analysis leveraging. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering, FormaliSE '18, p. 40–49. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3193992.3193994. URL `https://doi.org/10.1145/3193992.3193994`
24. Ferrara, P., Mandal, A.K., Cortesi, A., Spoto, F.: Cross-programming language taint analysis for the iot ecosystem. Electronic Communications of the EASST **77** (2019). DOI 10.14279/tuj.eceasst.77.1104
25. Ferrara, P., Negrini, L.: Sarl: Oo framework specification for static analysis. In: M. Christakis, N. Polikarpova, P.S. Duggirala, P. Schrammel (eds.) Software Verification, pp. 3–20. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-63618-0_1
26. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: Experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2021, p. 1–6. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3460946.3464316. URL `https://doi.org/10.1145/3460946.3464316`
27. Furr, M., Foster, J.S.: Polymorphic type inference for the jni. In: P. Sestoft (ed.) Programming Languages and Systems, pp. 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). DOI 10.1007/11693024_21
28. Furr, M., Foster, J.S.: Checking type safety of foreign function calls. ACM Trans. Program. Lang. Syst. **30**(4) (2008). DOI 10.1145/1377492.1377493. URL `https://doi.org/10.1145/1377492.1377493`
29. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: A. Bouajjani, O. Maler (eds.) Computer Aided Verification, pp. 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-02658-4_52
30. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: S. Chakraborty, J.A. Navas (eds.) Verified Software. Theories, Tools, and Experiments, pp. 1–18. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-41600-3_1
31. Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In: L. Hendren (ed.) Compiler Construction, pp. 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-78791-4_15
32. Kwon, J.: Tendermint: Consensus without mining (2014). Available at `https://tendermint.com/static/docs/tendermint.pdf`
33. Kwon, J., Buchman, E.: Cosmos whitepaper (2019). Available at `https://v1.cosmos.network/resources/whitepaper`

34. Lee, S., Lee, H., Ryu, S.: Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, p. 127–137. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3324884.3416558. URL `https://doi.org/10.1145/3324884.3416558`
35. Li, S., Tan, G.: Finding bugs in exceptional situations of jni programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, p. 442–452. Association for Computing Machinery, New York, NY, USA (2009). DOI 10.1145/1653662.1653716. URL `https://doi.org/10.1145/1653662.1653716`
36. Mell, P., Grance, T., et al.: The nist definition of cloud computing. National Institute of Science and Technology, Special Publication **800**(2011), 145 (2011)
37. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**(1), 31–100 (2006). DOI 10.1007/s10990-006-8609-1
38. Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native c extensions. In: C. Drăgoi, S. Mukherjee, K. Namjoshi (eds.) Static Analysis, pp. 323–345. Springer International Publishing, Cham (2021). DOI 10.1007/978-3-030-88806-0_16
39. Negrini, L.: A generic framework for multilanguage analysis. Ph.D. thesis, Universitá Ca' Foscari Venezia (2023)
40. Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings, p. 267–290. Springer-Verlag, Berlin, Heidelberg (2021). DOI 10.1007/978-3-030-67067-2_13. URL `https://doi.org/10.1007/978-3-030-67067-2_13`
41. Pinto, P., Carvalho, T., ao Bispo, J., Ramalho, M.A., ao M.P. Cardoso, J.: Aspect composition for multiple target languages using lara. Computer Languages, Systems & Structures **53**, 1–26 (2018). DOI 10.1016/j.cl.2017.12.003. URL `https://www.sciencedirect.com/science/article/pii/S147784241730115X`
42. Porru, S., Pinna, A., Marchesi, M., Tonelli, R.: Blockchain-oriented software engineering: Challenges and new directions. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 169–171 (2017). DOI 10.1109/ICSE-C.2017.142
43. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) **29**(5), 26—-es (2007). DOI 10.1145/1275497.1275501. URL `https://doi.org/10.1145/1275497.1275501`
44. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). DOI 10.1109/JSAC.2002.806121
45. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002). DOI 10.1145/514188.514190. URL `https://doi.org/10.1145/514188.514190`
46. Sharir, M., Pnueli, A., et al.: Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences . . . (1978)
47. Spoto, F.: The julia static analyzer for java. In: X. Rival (ed.) Static Analysis, pp. 39–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). DOI 10.1007/978-3-662-53413-7_3
48. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for java. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00, p. 264–280. Association for Computing Machinery, New York, NY, USA (2000). DOI 10.1145/353171.353189. URL `https://doi.org/10.1145/353171.353189`
49. Tan, G., Morrisett, G.: Ilea: Inter-language analysis across java and c. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07, p. 39–56. Association for Computing Machinery, New York, NY, USA (2007). DOI 10.1145/1297027.1297031. URL `https://doi.org/10.1145/1297027.1297031`
50. Teixeira, G., Bispo, J.a., Correia, F.F.: Multi-language static code analysis on the lara framework. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the

State Of the Art in Program Analysis, SOAP 2021, p. 31–36. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3460946.3464317. URL `https://doi.org/10.1145/3460946.3464317`

51. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: M. Hind, A. Diwan (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, pp. 87–97. ACM (2009). DOI 10.1145/1542476.1542486
52. Vongsingthong, S., Smanchat, S.: Internet of things: a review of applications and technologies. Suranaree Journal of Science and Technology **21**(4), 359–374 (2014)
53. Wei, F., Lin, X., Ou, X., Chen, T., Zhang, X.: Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, p. 1137–1150. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3243734.3243835. URL `https://doi.org/10.1145/3243734.3243835`