SootUp: A Redesign of the Soot Static Analysis Framework

Kadiray Karakaya[1], Stefan Schott[1], Jonas Klauke[1], Eric Bodden[1,2], Markus Schmidt[1], Linghui Luo[3]⋆, and Dongjie He[4]

[1] Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
{kadiray.karakaya,stefan.schott,jonas.klauke,eric.bodden,markus.schmidt}@upb.de
[2] Fraunhofer IEM, Paderborn, Germany
[3] Amazon Web Services, Berlin, Germany
llinghui@amazon.de
[4] University of New South Wales, Sydney, Australia
dongjieh@cse.unsw.edu.au

Abstract. Since its inception two decades ago, Soot has become one of the most widely used open-source static analysis frameworks. Over time it has been extended with the contributions of countless researchers. Yet, at the same time, the requirements for Soot have changed over the years and become increasingly at odds with some of the major design decisions that underlie it. In this work, we thus represent SootUp, a complete reimplementation of Soot that seeks to fulfill these requirements with a novel design, while at the same time keeping elements that Soot users have grown accustomed to.

Keywords: Static program analysis · Soot · SootUp.

1 Introduction

Soot is a program analysis framework for Java and Android. It has been popular in academia for prototyping novel static and dynamic analysis approaches, many of which have been published at international conferences [1, 3, 5, 6, 13, 14, 19, 20, 22, 28]. In 2000 [29], Soot was introduced as an optimization framework for Java. Back then, when just-in-time compilers were still in their infancy, ahead-of-time optimization of Java code was a major field of research. Over the years, the research community's interest has been dominantly shifting to static code analysis, for diverse purposes. Soot remained relevant due to some of its strengths, particularly its popular intermediate representations.

One of the core features of Soot is its main intermediate representation (IR), Jimple [30]. When seeking to perform program analysis on Java, both bytecode and source code are usually suboptimal representations to work with. Java bytecode represents a program to be *executed*, using a stack-based instruction set. Java source code, on the other hand, represents it on a higher level, using

⋆ The work was done prior to joining Amazon.

nested scopes and control-flow constructs for better *readability.* Soot's JIMPLE IR is a so-called three-address code representation [12] that combines the best of both worlds: It uses local variables instead of a stack. This simplifies data-flow equations because all values that an operation consumes or produces are readily accessible through its operands. It also uses explicit control flow without nesting, i.e., solely through conditional or unconditional gotos. In result, every Jimple instruction is atomic, there can be no nesting. Complex source-code statements, which perform multiple consecutive operations, e.g. a numerical computation with a subsequent cast, are broken down into multiple individual IR instructions. This enables the creation of simple control flow graphs (CFGs), which one can then use to analyze a method's control and data flow with relative ease.

Furthermore, SOOT offers multiple algorithms, with varying degrees of precision and complexity, for constructing call graphs. They resemble an essential data structure for performing inter-procedural static analysis, as it models how a program's methods call one another. For object-oriented programming languages like Java, call graph construction is particularly challenging. This is because in Java method calls are virtual by default, in which case their call target is dependent on an object's runtime type. A reference variable's declared type can only bound the possible call targets. To resolve call targets precisely one must compute all of the variable's possible runtime types. A popular way to do this is through pointer analysis. SOOT provides such call graph computation through its pointer analysis framework SPARK.

Over the years, SOOT has frequently been extended to incorporate new features, and, in doing so, even early on it became clear that some of its design decisions were suboptimal, yet hard to remedy after the fact. For instance, SOOT has always been all-around monolithic. It heavily uses the singleton design pattern, causing strong coupling, and it always sought to be both a command line tool and a library, causing sometimes conflicting views on who owns the thread of control. In SOOT, everything can be accessed and manipulated via the *singleton* "scene". This forbids keeping multiple scenes in memory, and any sensible parallelization. SOOT also contains many features that by now are considered obsolete, e.g. other barely used IRs and an outdated source-code frontend, which are hard to remove without breaking useful but *untested* functionality.

This paper presents SOOT's successor framework SOOTUP. With SOOTUP, we aim to keep the most important features of SOOT, yet to also overcome its major drawbacks. We designed SOOTUP as a *modular library.* This allows one to pick out the necessary modules for a specific use case. For instance, clients that only require bytecode analysis would add a dependency to the bytecode frontend module. This is possible due to SOOTUP's core module being a generic implementation that allows plugging in frontends for arbitrary programming languages. Instead of a singleton scene object, SOOTUP introduces the concept of *views*, where each view may hold a different version of the analyzed program, or different programs altogether. To enable safe parallelization and caching, the new Jimple IR is immutable by default, allowing instrumentation only at certain safe points. SootUp is open-sourced at GitHub.[1]

To summarize, this paper presents the following contributions:

- The design decisions behind SootUp's architecture that accommodate current research requirements,
- a demonstration of its new API, which aims for better usability,
- suggestions for Soot-based analysis tools on how to switch to SootUp, and
- the roadmap for further development of SootUp.

The remainder of this paper is organized as follows. In Section 2, we introduce the design decisions that shaped SootUp. In Section 3, we demonstrate the new API on example use cases. In Section 4, we list currently supported tools and discuss how to upgrade tools to use SootUp. In Section 5, we explain SootUp's development process and how one can contribute to it. We present the future work in Section 6, related work in Section 7 and conclude with Section 8.

2 Design Decisions

We next discuss the main design decisions that underly SootUp, and how they address some of the major shortcomings of Soot. We introduce the new architecture and excerpts of the new API.

2.1 Modular Architecture

SootUp's most notable architectural difference from its predecessor is the clear separation of its components into independent modules. Figure 1 shows its architectural overview. One of the goals of the new architecture is to allow SootUp to be used as a language-*independent* static analysis framework. It is not tightly coupled to any programming language. The most recent release (1.1.2) includes frontends for Java bytecode, Java source code and a now generic, i.e., language-independent form of Jimple. We delegate the language support to external frontend providers and expect them to *extend* the *generic* Jimple. This is a significantly different mechanism than Soot had offered for language support before. Previously, to analyze programs not in Java, one needed to convert their code to the (*Java-specific*) Jimple. With SootUp, instead one defines language-specific features by extending the core set of Jimple language constructs.

The *core* module encapsulates the main functionality based on the generic Jimple. It defines the Jimple language constructs such as expressions, constants and statements. The statements make up control-flow graphs (CFGs), which may be forward, backward, mutable or immutable. The CFGs are representations for the bodies of `SootMethod`s. `SootMethod`s constitute `SootClass`es, the backbone of SootUp's *core object model*. All of these objects are accessible through `View`s.

We have conceptualized the `View` as the main interface the user interacts with. In the case of a single view, this corresponds to the `Scene` object in Soot.

[1] https://github.com/soot-oss/SootUp/

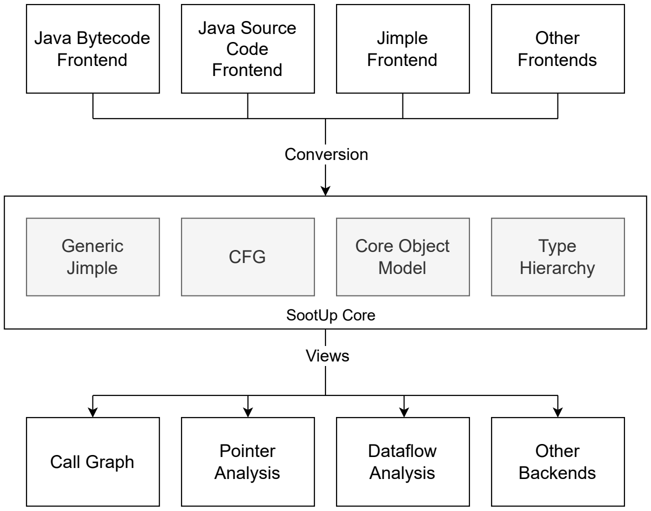

Fig. 1. Overview of SootUp's Architecture. White boxes are Java modules.

Because of the `Scene`'s singleton nature, running multiple analyses simultaneously was virtually impossible in Soot [15]. SootUp overcomes this drawback by allowing as many `Views` as desired to co-exist.

Additionally, SootUp comes with a new extensible *Call Graph* framework. It allows plugging in arbitrary strategies for resolving virtual method dispatches. These strategies could vary, for instance, to optimize the precision or scalability, which are often tweaked using different *Pointer Analysis* algorithms. Interprocedural *Dataflow Analysis* is one of the most successful methods for detecting bugs and security vulnerabilities. SootUp supports out-of-the-box context-sensitive data-flow analysis using the popular Heros [4] dataflow analysis framework.

2.2 On-Demand Class Loading

While Soot loads all `SootClass`es that are referenced in a currently resolving `SootClass`, SootUp is designed with a layer of indirection. SootUp makes use of identifiers to reference actual, possibly already loaded, instances of a respective `SootClass` and stores those identifiers that reference other `SootClass`es, `SootMethod`s or `SootField`s. This decreases unnecessary computations of unused `SootClass`es, i.e. those which are referenced but whose contents are not of interest. Doing so, additionally, enables parallel class loading. Because the loading of a class does not depend on the loading of the classes that it references, each class can be loaded independently. As a side effect, it renders the concept of *phantom classes*, known from Soot, obsolete, as its purpose is to create a facade `SootClass` in case of missing a class definition of a referenced `SootClass`. This case is now cleanly handled by the `View`, which simply returns no further information.

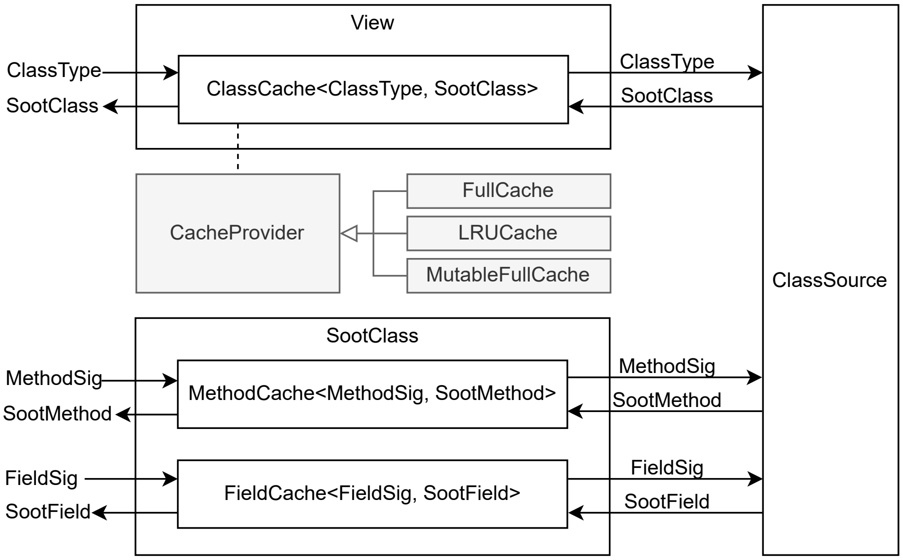

Fig. 2. SootUp's On-Demand Class Loading Mechanism

Figure 2 models SootUp's new on-demand class loading mechanism. The `View` is the central access point that streamlines the resolving and caching process. The caching strategy can be configured by using one of the cache providers. `FullCache` is the default option, which suffices in most cases where the cache does not need to be freed. Alternatively `LRUCache` manages the cache based on the least recent use and `MutableFullCache` gives the control of the cache to the client. After obtaining a `SootClass`, by querying it with its unique identifier (`ClassType`) from the `View`, one can obtain its `SootMethods` and `SootFields` that are cached within the `SootClass`.

2.3 Focus on an Intuitive API

Soot's users often complain about a lack of documentation. Its issue tracker is filled with "how to"[2] questions. We believe the underlying problem is, primarily, its complicated API design. Based on our past experience, when developing SootUp, an intuitive API design has always been strongly in focus.

Figure 3 shows the process of setting up a `Project`, creating a `View` and accessing a `SootMethod` object. First, users create an `AnalysisInputLocation` that points to a target program's path. Second, they create a `Project` by specifying the target language. The `Project` can be used to create a `View`. At this point, the `View` knows where the target program is located and which language frontend needs to be used to load its classes.

[2] https://github.com/soot-oss/soot/issues?q=how+to+in%3Atitle

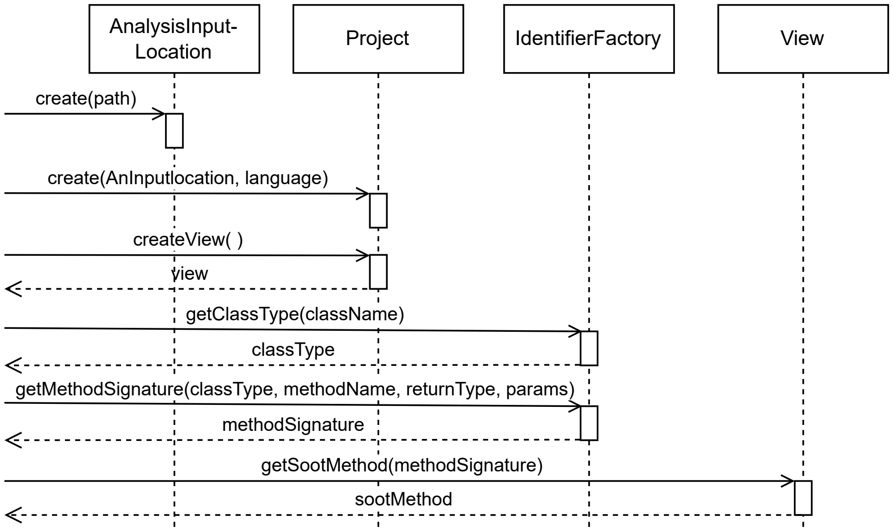

Fig. 3. SootUp's API for Creating a View and Accessing a SootMethod

The `View` loads the elements of the target program only when they are queried, and memoizes them through configurable caching providers enabled by the new immutable IR design. The memoization is fine-grained, it works at the level of field, method, interface and modifier definitions. SootUp can create references to all of these objects via a corresponding language-specific `IdentifierFactory`. The references, i.e., the identifiers, are then used to access the queried elements of the target program.

Class types and signatures (for methods and fields) are considered global identifiers, across possibly concurrent instances of Projects and Views. They are created and pooled by the singleton instance of `IdentifierFactory` to reduce memory consumption. Additionally, it is cheaper to invoke `hashCode()` and `equals()` on the identifiers than on the IR objects that the identifiers reference.

2.4 Library by Default

Soot had always been designed to be a standalone CLI (command-line interface) tool. This meant that it was expected to own the thread of control, which often hindered a tight integration of Soot into integrated development environments (IDEs) or CI/CD pipelines, which are themselves frameworks and expect to own the thread of control as well. Also, a CLI aggregates all of the underlying functionality and makes it accessible via a single channel. This requires bundling everything together and contradicts our goals of providing lean modules.

To avoid this, we have conceptualized SootUp as a library by default. In SootUp, clients can depend on individual modules. For instance, to access the CFGs of a compiled program's methods, one needs to add a dependency to the

Java Bytecode Frontend and *Core* modules. Further module dependencies can gradually be added later on when needed.

The library nature allows *the clients* to own the thread of control. This is preferable, especially, when using SootUp for other purposes than program analysis, or when using it as part of other analysis frameworks. SootUp also provides rather sophisticated functionality as a framework, with inversion of control, for instance when building call graphs or performing dataflow analyses.

Yet, SootUp is not quite stateless. As shown in Figure 3, the state is managed mainly by the `IdentifierFactory` and `View`. `View` *instances* keep references to all the memoized objects, they are not garbage collected unless the client releases the reference to the `View`. `IdentifierFactory`, on the other hand, maintains the global state of unique identifiers statically. It is the only singleton in SootUp, which might be shared across different views. In other words, if the client terminates then only the state in the `IdentifierFactory` will be retained.

2.5 Immutable IR by Design

Soot was designed as a program optimization tool. Its main purpose was to enable the analysis and transformation of method bodies. As the research trend has shifted from program optimization to program analysis, we believe there is limited use in still maintaining mutable objects in a mutable IR.

Mutable objects are not easily shared between several entities. One needs to constantly account for unintended changes. They very much complicate parallelization at any level. To counter this problem, we have designed SootUp's Jimple IR to be immutable by default. This assures that there are no accidental modifications and that values can be safely shared and cached.

```
class Body {
  ...

  Body withStmts(List<Stmt> stmts) {
    return new Body(stmts);
  }
}
```

Listing 1.1. Modifying a Method Body via Withers

To ensure immutability we have slightly adjusted the API as well. Many classes do not have *setters* anymore, they have *withers* instead. Withers still allow modifications via new object copies with modified properties. Listing 1.1, for instance, shows how one can still modify the statements of a method body.

2.6 Changes to Jimple

Originally, Jimple was designed to be an IR for program optimization to fit Soot's primary use case. Since the purpose of SootUp has been shifted towards program analysis instead of optimization, we adjusted the Jimple IR towards this purpose. For efficiency reasons, a Java compiler compiles any `switch` statement to either a `tableswitch` or a `lookupswitch` bytecode instruction. Since the distinction is needed to transform the optimized Jimple back to bytecode, Jimple also made a distinction between `tableswitch` and `lookupswitch` statements. However, virtually all program analyses will treat both kinds of statements identically. Because of this, in SootUp both statements have been merged into a single `switch` statement, simplifying analysis implementations.

Another novelty in SootUp's Jimple is the added support for language extensibility. SootUp is designed to be an analysis framework that not just supports Java, but also other programming languages as well. To allow for this multi-language support, a basic Jimple IR has been implemented in a generic way that allows for easy extension with language-specific features. For the Java implementation, we extended this basic Jimple IR with import statements and annotations, two features that are highly specific to the Java language. Annotations are supported by extending Jimple's class type definition. Just like in Java source code, import statements improve the readability of Java-Jimple statements. Java-Jimple now allows referring to simple class names by defining their fully qualified names as imports. Likewise, basic Jimple can be extended to support features specific to other languages, e.g. JavaScript or Python.

3 Demonstration

In Section 2.3, we provided a glimpse of the new API. In this section, we demonstrate the new API with a set of most common use cases.

3.1 Setup

The code snippet in Listing 1.2 shows the starting point in SootUp to build an analysis project. The project builder requires two inputs: (1) the language of code to be analyzed and its version, as SootUp supports multiple languages; (2) the location of the analysis target. In this example, we are setting the analysis language as Java with version 8 and adding a Java classpath analysis input location that points to the analysis target. Note that one can add multiple analysis input locations to the project builder. The Java bytecode frontend accepts any of the Java archive formats (JAR or WAR), Android packages (APK), ZIPs or individual `.class` files. The Java source and the Jimple frontends accept `.java` and `.jimple` files respectively. To resolve a given class, the view will inspect all of the given analysis input locations.

```java
JavaLanguage language = new JavaLanguage(8);
JavaProject project = JavaProject.builder(language)
  .addInputLocation(new JavaClassPathAnalysisInputLocation("/path"))
  .build();
JavaView view = project.createView();
```

Listing 1.2. The creation of a view in SootUp

3.2 Obtaining a Method Body

Assume the target code example in Listing 1.3. Following the API usage in Section 2.3, next we need to obtain a reference to the target class. To do so, as shown in Listing 1.4, we get the `IdentifierFactory` from the view at line 6. We obtain the target class type at line 7 and likewise the target method's signature at line 8. A class is rather straightforward to identify, i.e. with a string corresponding to its *fully qualified name*, e.g. `"org.example.Main"` in this example.

Identifying methods requires a bit more information, as one needs to specify its containing class type, name, return type and parameter list to uniquely identify it. In this example, we use the target class type (`ct`) that we have created, set the name as `"run"` and return type as `"void"`. It is important to refer to any class type with its fully qualified name. For instance, while in Java it suffices to write `String[] args` to define the parameters as a string array, SootUp needs the definition as `java.lang.String[]`.

```
package org.example;
public class Main {

  void run(String[] args) {
    ...
  }
}
```

Listing 1.3. Target code example

```
6  IdentifierFactory factory = view.getIdentifierFactory();
7  ClassType ct = factory.getClassType("org.example.Main");
8  MethodSignature mSig = factory.getMethodSignature(
9    ct, "run", "void", Collections.singletonList("java.lang.String[]"));
```

Listing 1.4. Definition of a class type and a method signature using SootUp

The method signature that we created (`mSig`) can now be used to query the actual method object from the view. This is shown at line 10 in Listing 1.5. As the new API follows the modern Java best practices, `view.getMethod()` returns an *optional*, at line 11, we therefore test this optional for its presence and obtain the methods body. At line 12, we output all the statements of the method.

```
10  view.getMethod(mSig)
11    .ifPresent(method ->method.getBody()
12      .getStmts().forEach(System.out::println));
```

Listing 1.5. Output all statements in a method body using SootUp

3.3 Call Graph Generation

A call graph models the calls between the methods of a target program, which makes it an essential data structure when performing interprocedural program analyses. SootUp's new call graph framework is based on a generic notion of a *CallGraphAlgorithm*, which can be extended by specific call graph algorithm implementations. The call graph algorithms only need to specify how they *resolve* a call. Resolving can be based on the static class hierarchy (e.g. CHA [7], RTA [2]) or based on sophisticated pointer analyses [16].

```
13    CallGraphAlgorithm cha = new ClassHierarchyAnalysisAlgorithm(view);
14    CallGraph cg = cha.initialize(Collections.singletonList(mSig));
15    cg.containsMethod(anotherMethod)
16    cg.callsFrom(mSig)
```

Listing 1.6. Call graph generation using SootUp

Listing 1.6 shows an example of call graph generation using the new API. Since the view maintains all the classes and methods, it needs to be passed to the call graph algorithm, e.g. the `ClassHierarchyAnalysisAlgorithm` at line 13. The call graph algorithm is initialized at line 4, by specifying the entry method, which returns a `CallGraph` object. The call graph can be queried for method reachability, e.g. at line 14, or can be iterated by retrieving the calls from the entry method, e.g. at line 15.

3.4 Body Interceptors

Body interceptors in SootUp replace the concept of transformers in Soot. They essentially allow modifying method bodies, for instance, to add, remove or replace statements. As with the other objects, methods are immutable by default. Therefore, in SootUp any modifications to the method body must be performed during the body-building phase.

```
ClassLoadingOptions clo = new ClassLoadingOptions() {
  @Override
  public List < BodyInterceptor > getBodyInterceptors() {
    return Collections.singletonList(new DeadAssignmentEliminator());
  }
};
JavaView view = project.createView(analysisInputLocation -> clo);
```

Listing 1.7. Specifying Body Interceptors

Listing 1.7 shows an example of specifying a body interceptor. In this example the `DeadAssignmentEliminator` is specified. The body interceptors must be defined as part of the class loading options, as they are applied during class loading. The options are passed during the view creation.

4 Tool Support

Soot-based tools can be upgraded to use SootUp instead, however, depending on their implementation, the upgrading effort may vary. We next present the tools that SootUp currently supports and provides as submodules. We also suggest the roadmap for Soot-based tools for switching to SootUp.

4.1 Heros

Heros [4] enables defining interprocedural dataflow analysis using the IFDS (interprocedural, finite, distributive subset) [23] and IDE (inter-procedural distributive environments) [24] conceptual frameworks. Both frameworks reduce dataflow analysis problems to graph reachability. While IDE well suits the analysis problems with large domains (such as typestate or constant propagation analysis), IFDS is the primary choice for reachability analyses with a small domain (e.g. taint analysis).

```
1  JimpleBasedInterproceduralCFG icfg =
2          new JimpleBasedInterproceduralCFG(view, entryMethod);
3
4  IFDSTaintAnalysisProblem problem =
5          new IFDSTaintAnalysisProblem(icfg, entryMethod);
6
7  JimpleIFDSSolver<?, InterproceduralCFG<Stmt, SootMethod>> solver =
8          new JimpleIFDSSolver(problem);
9
10 solver.solve();
```

Listing 1.8. IFDS analysis using Heros

SootUp provides the Heros framework within its analysis submodule. Listing 1.8 shows an example on running an IFDS analysis using Heros. SootUp implements Heros' `InterproceduralCFG` interface with the Jimple-specific `JimpleBasedInterproceduralCFG`. To instantiate it, the client needs to pass the view and an entry method as shown at line 1. Heros defines IFDS problems as an abstract class with `DefaultIFDSTabulationProblem`, this is extended by `DefaultJimpleIFDSTabulationProblem` in SootUp. However, the clients still need to define their custom IFDS analyses with problem-specific lattices, flow-functions and merge operators. An example of a basic IFDS-based taint analysis problem is available in SootUp, which is instantiated at line 4. SootUp extends Heros' generic `IFDSSolver` with the `JimpleIFDSSolver` by concretizing it with `Stmt` (equivalent to `Unit` in Soot) and `SootMethod`.

4.2 Qilin

Pointer information is an integral part of precise program analyses. Soot's pointer analysis frameworks, Spark [16] and its context-sensitive alternative Paddle [17], have been popular in academia, as they provide a solid ground for researching novel algorithms. As we observe, however, the research trend is moving towards more sophisticated approaches with increased pointer analysis precision. For instance, context-sensitivity can be applied *selectively* rather than uniformly across the whole program [18].

Qilin [11] is a state-of-the-art flow-insensitive Java pointer analysis framework that was recently designed for supporting fine-grained selective context sensitivity while subsuming existing traditional method-level context sensitivity as a special case. Since Qilin is fully written in Java and operates on the Jimple IR of Soot, we were able to seamlessly incorporate Qilin into SootUp as a submodule with only minor engineering efforts. Qilin supports a rich set of pointer analyses such as Andersen's context-insensitive analysis as implemented in Spark [16], k-limiting callsite-sensitive analysis [26], k-limiting object-sensitive analysis [21,27], and other recent advancements in pointer analysis. By providing Qilin as a SootUp submodule, we aim to foster comparative research using a broader set of pointer analysis algorithms.

```
1 PTAPattern ptaPattern = new PTAPattern("2o");
2 Collection entries = Collections.singleton(mainSig);
3 PTA pta = PTAFactory.createPTA(ptaPattern, view, entries);
4 pta.run();
5 CallGraph cg = pta.getCallGraph();
```

Listing 1.9. Call graph generation using a pointer analysis in Qilin

Listing 1.9 gives an example of 2-object sensitive pointer analysis using Qilin. In lines 1 and 2 the flavor of pointer analysis is specified and the entry method is set. In line 3 an instance of 2-object sensitive analysis is created which is subsequently executed in line 4. As the pointer analysis in Qilin supports on-the-fly call graph construction, the resulting call graph is retrieved in line 5. In addition, pointer analysis in Qilin provides the APIs `reachingObjects`, for computing the points-to set of any variable and `mayAlias`, for checking whether two variables are aliases.

4.3 Roadmap for Other Soot-based Tools

SootUp is not a drop-in replacement for Soot. It is essentially a complete rewrite with a new architecture and API. We therefore primarily recommend SootUp to be used for new projects. However, existing tools that are based on Soot can be upgraded to SootUp with some effort. The SootUp team has been working on upgrading some Soot-based tools to SootUp. So far, we see that the roadmap, and thus the effort, for a specific tool to upgrade to SootUp will differ heavily based on how it is implemented. We have been seeing three recurring patterns: (1) generic tools that do not directly depend on Soot, (2) tools that depend on Soot but work with their own domain objects, (3) tools that depend on Soot and work directly with Soot objects.

Generic tools can swiftly be upgraded to SootUp. For instance, the API of the Heros solver provides interfaces based on Java generics. Its interfaces can be extended with concrete tool-specific objects. The only requirement for SootUp to use the IFDS solver was to extend necessary interfaces by providing SootUp-specific objects.

Upgrading tools that use their own domain objects to SootUp is also simple. For instance, Boomerang [28] and SparseBoomerang [14], state-of-the-art demand-driven pointer analysis frameworks, implement their core functionality within their own domain objects that correspond to classes, methods and statements. These tools require SootUp's objects to be converted to their domain objects via implementing an adapter.

Upgrading tools that work directly with Soot objects is a more complex task. FlowDroid [1], a popular Android information flow analysis tool, is highly intertwined with Soot. It is hard to determine where exactly the boundaries of FlowDroid are and how to separate it from Soot. Therefore, at this point, we anticipate that FlowDroid and tools of similar nature need a major rewrite to upgrade to SootUp. Nonetheless, we are considering upgrading even FlowDroid to SootUp in the future.

5 Development

We next explain SootUp's development process, and how one can extend or contribute to SootUp.

5.1 SootUp's Development Process

We have incepted SootUp as a greenfield project. This choice not only granted us more freedom to restructure its architecture but also to employ a more modern software development process. Our new development process centers around continuous quality assurance. Soot lacked proper test coverage, which complicated adding new features or any kind of nontrivial refactoring. To overcome this, we made testing an integral part of SootUp from the very beginning. SootUp is loaded with exhaustive unit and regression tests. We continuously observe its test coverage and enforce newly added code to maintain the same level of coverage. To ensure that no new feature breaks or unintendedly changes SootUp's behavior, tests are executed for every new commit to SootUp's code repository through a continuous integration pipeline.

We seek to make SootUp more accessible to everyone. Our focus on an intuitive API design, as we explained in Section 2.3, is the first step in this direction. Further, we prioritize documentation and make it part of the development process. Our public-facing API elements are required to have Javadoc. Yet, we have learned, considering the questions in Soot's issue tracker, that Javadoc alone is not enough. We thus maintain a documentation page[3] to elaborate on some of the main concepts of SootUp's usage and provide more insight. To make the documentation beginner-friendly, we demonstrate the most common use cases with supporting code examples. From experience, we know that documentation tends to fall behind the most recent development state. To prevent this, we maintain the example code as part of SootUp's code repository. By doing so we ensure that the example code always compiles and functions with the most recent state.

SootUp is currently published at Maven Central. We have announced the first release (`v1.0.0`) in December 2022. Since then, we have been frequently releasing new features and bug fixes, the most recent version at the time of writing is `v1.1.2`, published in June 2023. While, due to existing tool dependencies,

[3] https://soot-oss.github.io/SootUp/

Soot and SootUp will coexist for a while, the bulk of our maintenance efforts will henceforth be directed toward SootUp rather than Soot.

5.2 Extending and Contributing to SootUp

Concerning community engagement, SootUp will follow in the footsteps of Soot. While SootUp's development is currently still carried by Paderborn University, we are open for others to join the team. The main motivation behind our development efforts until the first release was to realize the design decisions laid out in Section 2. Since the first release, we have been focusing more on community feedback, such as bug reports and feature requests. Just like its predecessor, we expect SootUp to be shaped around the needs and contributions of the research community. We are eager to incorporate external contributions and very much welcome feature and pull requests. Repeat contributors may become core development team members with full commit rights.

To maintain an active community, we set up a discussion board on GitHub. This allows the community to participate in Q&As, suggest new ideas or simply discuss in an informal setting. SootUp is open-sourced with a GNU General Lesser Public License v2.1 (LGPL-2.1) [10]. It allows SootUp to be modified as long as the modifications are stated and licensed under the same license.

6 Future Work

SootUp is set to be the successor of the old Soot framework. Soot has been developed and improved for more than 20 years, so there are still multiple analysis utilities that need to be adapted to SootUp. Furthermore, we aim to keep up with advancements in the field of static program analysis and implement support for better callgraph construction approaches and more precise pointer-analysis techniques in SootUp as they are developed.

Being able to analyze Android applications was one of the main reasons for Soot's popularity. SootUp currently allows one to analyze Android applications with the help of dex2jar.[4] This is an interim solution, as dex2jar is no longer actively maintained. In the meantime, we are working on a more robust solution based on Dexpler [3].

SootUp was designed with extensibility for other programming languages in mind. To allow for cross-boundary program analyses, we aim to implement new frontends for other languages. We especially aim at implementing a Python and a JavaScript frontend, due to the popularity of these languages.

Another goal for SootUp is to provide a means to enable the analysis of partial programs. To process an uncompiled Java source code project using Soot or SootUp, the whole code base of the project, alongside all its dependencies, needs to be available either during compilation or during processing with the source code frontend. However, in some scenarios only part of the code base is

[4] https://github.com/pxb1988/dex2jar

available. In the future, we aim to provide support for processing such partial programs. By being able to generate Jimple from only partially available source code and substituting the missing information with either data that can be inferred from whatever is available of the code base or providing a means to additionally specify missing parts.

In the future, we plan to also perform more evaluations in regards to SootUp. An API design that is as intuitive as possible for its users was one of the primary considerations when designing SootUp. To validate the API design, we plan to perform user studies with various types of user groups like researchers and software developers. Furthermore, we plan to benchmark SootUp's performance and compare it against other analysis frameworks and especially its predecessor.

7 Related Work

Apart from Soot, there are various research-oriented static analysis frameworks. The most notable ones for Java are WALA [31], Doop [5] and OPAL [9]. WALA enables analyzing multiple programming languages such as Java, Javascript, and recently also Python [8]. It focuses on efficient static analysis by using specialized data structures. WALA's IR is close to JVM bytecode, but in contrast, it is based on SSA (static single assignment). Instead of operand stacks, it uses symbolic registers. SootUp is currently integrated with WALA's source code frontend, which enables SootUp to support source code in the same capacity as WALA does. Doop was originally developed as a pointer analysis framework. It enables defining static analyses declaratively and uses a Datalog solver. Doop's IR is also based on Jimple. It could probably be upgraded to SootUp with minor effort. OPAL provides highly configurable static analysis using abstract interpretation. Another notable static analysis framework is PhASAR [25], a framework that enables static analysis for C and C++ applications through the LLVM IR.

8 Conclusion

We have presented SootUp, a complete overhaul of the popular Soot optimization and analysis framework for Java. SootUp shifts the purpose from optimization to static code analysis and fully modernizes the original Soot implementation. SootUp implements all the lessons learned from the last 20+ years of development and usage of the original Soot framework. It comprises many improvements like a new user-centric API, a fully parallelizable architecture and an new variant of the Jimple intermediate representation offering extensibility for multi-language support. With all these changes and improvements in place, SootUp aims to be a worthy successor of the good old Soot framework and to enable the implementation of modern Java code analyses.

Acknowledgements. The development of SootUp was supported by the Research Software Sustainability funding line of German Research Foundation (DFG), and the Heinz Nixdorf Institute (HNI).

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)
2. 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. pp. 324–341 (1996)
3. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. pp. 27–38 (2012)
4. Bodden, E.: Inter-procedural data-flow analysis with ifds/ide and soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. pp. 3–8 (2012)
5. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. p. 243–262. OOPSLA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1640089.1640108, https://doi.org/10.1145/1640089.1640108
6. Dann, A., Hermann, B., Bodden, E.: Sootdiff: Bytecode comparison across different java compilers. In: Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis. pp. 14–19 (2019)
7. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9. pp. 77–101. Springer (1995)
8. Dolby, J., Shinnar, A., Allain, A., Reinen, J.: Ariadne: analysis for machine learning programs. In: Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 1–10 (2018)
9. Eichberg, M., Hermann, B.: A software product line for static analyses: The opal framework. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. p. 1–6. SOAP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2614628.2614630, https://doi.org/10.1145/2614628.2614630
10. Free Software Foundation, I.: Gnu lesser general public license v2.1 - gnu project - free software foundation. https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html (1999), (Accessed on 10/09/2023)
11. He, D., Lu, J., Xue, J.: Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 30:1–30:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.30, https://drops.dagstuhl.de/opus/volltexte/2022/16258
12. Hoe, A.V., Sethi, R., Ullman, J.D.: Compilers—principles, techniques, and tools (1986)

13. Karakaya, K., Bodden, E.: Sootfx: A static code feature extraction tool for java and android. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 181–186. IEEE (2021)
14. Karakaya, K., Bodden, E.: Two sparsification strategies for accelerating demand-driven pointer analysis. In: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 305–316. IEEE (2023)
15. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (Oct 2011), https://www.bodden.de/pubs/lblh11soot.pdf
16. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) Compiler Construction. pp. 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
17. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM Transactions on Software Engineering and Methodology (TOSEM) **18**(1), 1–53 (2008)
18. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: A principled approach to selective context sensitivity for pointer analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**(2), 1–40 (2020)
19. Li, Y., Tan, T., Zhang, Y., Xue, J.: Program tailoring: Slicing by sequential criteria. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
20. Luo, L., Bodden, E., Späth, J.: A qualitative analysis of android taint-analysis results. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 102–114. IEEE (2019)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (jan 2005). https://doi.org/10.1145/1044834.1044835, https://doi.org/10.1145/1044834.1044835
22. Piskachev, G., Krishnamurthy, R., Bodden, E.: Secucheck: Engineering configurable taint analysis for software developers. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 24–29. IEEE (2021)
23. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61 (1995)
24. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science **167**(1-2), 131–170 (1996)
25. Schubert, P.D., Hermann, B., Bodden, E.: Phasar: An inter-procedural static analysis framework for c/c++. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 393–410. Springer (2019)
26. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, chap. 7, pp. 189–234. Prentice-Hall (1981)
27. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 17–30. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926390, https://doi.org/10.1145/1926385.1926390

28. Späth, J., Nguyen Quang Do, L., Ali, K., Bodden, E.: Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
29. Vallée-Rai, R., Gagnon, E.M., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing java bytecode using the soot framework: Is it feasible? In: International Conference on Compiler Construction (2000)
30. Vallee-Rai, R., Hendren, L.J.: Jimple: Simplifying java bytecode for analyses and transformations. Tech. rep., Technical report, McGill University (1998)
31. WALA: wala/wala: T.j. watson libraries for analysis, with frontends for java, android, and javascript, and may common static program analyses. https://github.com/wala/WALA, (Accessed on 10/04/2023)