TACAI: An Intermediate Representation Based on Abstract Interpretation

Michael Reif, Florian Kübler, Dominik Helm,
Michael Eichberg, Mira Mezini
Technical University of Darmstadt
Germany
<reif,kuebler,helm,mezini>@cs.tu-darmstadt.de

Ben Hermann
Universität Paderborn
Germany
ben.hermann@upb.de

Abstract

Most Java static analysis frameworks provide an intermediate presentation (IR) of Java Bytecode to facilitate the development of static analyses. While such IRs are often based on three-address code, the transformation itself is a great opportunity to apply optimizations to the transformed code, such as constant propagation.

In this paper, we propose TACAI, a refinable IR that is based on abstract interpretation results of a method's bytecode. Exchanging the underlying abstract interpretation domains enables the creation of various IRs of different precision levels. Our evaluation shows that TACAI can be efficiently computed and provides slightly more precise receiver-type information than Soot's Shimple representation. Furthermore, we show how exchanging the underlying abstract domains directly impacts the generated IR.

Keywords: three-address code, static single assignment, static analysis, Java bytecode

1 Introduction

To ease the implementation of static analyses, common static analysis frameworks for Java bytecode transform the stack-based bytecode into a three-address-code-based intermediate representation (TAC). As TAC representations have a much smaller instruction set than the original bytecode, they can ease the development of static analyses. Using such a transformation does not only remove the bytecode's operand stack—which complicates static analysis—but also enables the immediate application of optimizations [5, 14], e.g., constant propagation or dead path removal. As Bodden [1] observed, efficient static analyses with higher precision also yield better performance in subsequent analyses, emphasizing the need for optimizations. However, the optimizations' effect on subsequent analyses is still not well understood; in particular for optimizations related to the base representation on which subsequent analyses are built. This work explores the impact of the employed base representation on the call graphs built on top of different optimizations.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Despite that all major analysis frameworks use a typed base representation, they offer different precision. Most of Soot's [13] analyses use Jimple [14], an optimized TAC representation applying different optimizations during its generation (e.g. constant propagation), DOOP [11] uses Soot's Shimple[1] which is Jimple transformed to static single assignment (SSA) form, and WALA's [9] IR[2] is in an extended SSA [3] form which additionally captures local path conditions in π nodes [2]. In this paper we extend OPAL [7], formerly operating on bytecode directly, by a SSA-like representation without ϕ statements. However, compared to the other representations, ours optionally provides advanced type information about reference types using union and intersection types.

Initial observations reveal that especially Jimple, WALA IR, and TACAI apply different optimizations during the transformation such that the available type information varies in precision.

```
1 Collection c;
2 if(cond){ c = new ArrayList(); } else { c = new Vector(); }
3 c.add(null); // Call site
```

Listing 1. Precision Example

With the help of Listing 1, the differences between Jimple, WALA IR, and TACAI can be explained. Considering the call site at line 3, all three IRs provide type information with different precision. Whereas WALA IR only provides `c`'s declared type `Collection`, Jimple encodes the upper-type bound `List`, i.e., the common supertype of `ArrayList` and `Vector`. TACAI provides a union type of `ArrayList` and `Vector`—the most precise type information if `cond` is unknown. Hence, even a simple class hierarchy analysis call graph [4] already differs when these different IRs are used.

This paper reports on the design and implementation of TACAI, an abstract-interpretation-based intermediate representation with exchangeable abstract domains. We implemented TACAI as an extension of OPAL. Being able to adapt a domain provides the advantage that the precision of the information encoded by the TAC can easily be changed, e.g.,

[1] https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple, 03-04-2019.
[2] https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR), 03-04-2019.

to provide more precise information regarding the concrete type of receiver objects, integer values, and strings. We will discuss and evaluate how switching the used abstract domains affects the bytecode-to-TACAI transformation. Furthermore, we compare TACAI to Shimple, a well-established intermediate representation provided by Soot.

2 Approach

TACAI, our approach for a three-address code (TAC) representation, is based on the results of an intra-procedural abstract interpretation (AI) of a method's bytecode. This features two main properties: First, it enables intermediate-representation (IR) derivation at different precision levels by exchanging the underlying domains. Second, all information is computed at the same time in one step which offers improved performance when compared to classical compiler frameworks which typically compute comparable information in a step-wise manner [10]. In this step-wise approach collected information is oftentimes not shared between steps and, therefore, recomputed to reduce dependencies. While performing the AI, OPAL always computes the method's control-flow graph (CFG) and def-use/use-def information on-the-fly. Therefore, the CFG and def-use information immediately benefit from better domains and lead to simpler and less IR code. The CFG and def-use information are also made explicit in TACAI.

We reuse OPAL's domains starting with those operating at the type level and which will lead to an IR that has similar precision as Soot's Shimple representation. However, OPAL also provides domains that enable constant propagation and constant folding for primitive types. For reference values, domains are available which, for instance, precisely track the nullness, provide must-alias information, compute intersection and union types, or resolve local `Class.forName` calls. Using these domains enables the computation of an more precise IR when compared to typical IRs offered by the other frameworks. Furthermore, it is possible to tailor the precision at a very fine-grained level to a client's needs.

OPAL uses Scala's mixin-composition to configure the AI and to implement the semantics for the different sets of instructions. The default, namely $TACAI^{L0}$, performs all operations at the type level and is shown in Listing 2.

The semantics for each set of closely related instructions is implemented by one specialized trait. OPAL provides one trait for `integer`, `long`, `float`, and `double` based computations, one for method invocations, one for field accesses, and one for reference-value-based operations. The latter handles, e.g., `instanceof` checks, casts, and tests against `null`. Interactions between the traits are facilitated by requiring the implementation of a shared set of query methods. For example, every implementation that handles reference values has to implement the globally defined method to test if a value is `null`. The result of these methods is typically

```
1  trait TypeLevelDomain extends Domain
2      with DefaultReferenceValuesBinding
3      with DefaultTypeLevelIntegerValues
4      with DefaultTypeLevelLongValues
5      with TypeLevelLongValuesShiftOperators
6      with TypeLevelPrimitiveValuesConversions
7      with DefaultTypeLevelFloatValues
8      with DefaultTypeLevelDoubleValues
9      with TypeLevelFieldAccessInstructions
10     with TypeLevelInvokeInstructions
```

Listing 2. Example `TypeLevelDomain` Configuration

a three-state answer: Yes, No, or Unknown. For example, the method returning a reference value's nullness is used by the domain which is responsible for handling method calls. The latter checks—for each method invocation—if the receiver is `null`. If the receiver is known to be `null` the target method is not invoked, but a `NullPointerException` will be thrown instead. If the answer is `Unknown` the behavior can further be configured such that only the call is considered or additionally an exception is considered to be thrown.

Besides the $TACAI^{L0}$ configuration, two further configurations for a more precise TAC are preconfigured. In the first one ($TACAI^{L1}$), the `DefaultReferenceValuesBinding` (Line 2) is exchanged for an implementation that computes intersection and union types as well as must-alias information for reference values. Furthermore, special support for calls of the native method `System.arraycopy` is provided which checks for the non-nullness of the arrays and also validates the range that is to-be-copied. If this validation fails, appropriate exceptions are thrown which have to be correctly represented.[3] Lastly, constant folding and propagation is performed for integer values by exchanging the `DefaultTypeLevelIntegerValues` (Line 3) domain. The latter is required to identify `if` statements where the conditions evaluate to constant values and are therefore useless.

The most precise configuration ($TACAI^{L2}$) builds on top of $TACAI^{L1}$ and additionally performs method inlining for monomorphic calls. This is, e.g., useful for builders (e.g. StringBuilder/StringBuffer) which provide a fluent interface, enabling call chaining by always returning the current instance. In such cases, it is then possible to determine that all calls actually happen on the same instance. For that, Scala's stackable trait pattern is used to adapt the handling of method invocations, i.e., an additional trait is configured.

Table 1 shows the respective TACAI output for method `m` (cf. Listing 3) at all three levels. $TACAI^{L0}$ almost directly reflects the bytecode: The type of the variable `p1` (Line 2) is considered to be `Cloneable` after the cast operation. The code also contains the (useless) reference comparison (Line

[3] Special handling is provided for `System.arraycopy` because it is by far the most widely used native method in the JDK.

7), comparing the newly created `StringBuffer` (Line 4) with the reference returned by the `append` call (Line 6). $TACAI^{L1}$ is able to correctly identify that `p1`'s type is `Serializable with Cloneable`. This intersection type significantly restricts the set of subtypes when compared to the previous version. Additionally, both `p1` and `lv4` are found not to be null: `p1` because of the explicit nullness check (Line 0), the second because it is freshly allocated (Line 4). This guarantees that the invocations on `p1` (Line 3) and `lv4` (Lines 6 and 10) will not cause `NullPointerExceptions`. Although the chosen domain is able to track must-alias information intra-procedurally, the (useless) reference comparison is still found in the TAC. The identification of the must-alias relation in this case requires to know that the value returned by `append` is the self-reference `this`. By performing inlining, as done when computing the $TACAI^{L2}$, this information becomes available and, therefore, the useless comparison can be removed and subsequently, the `if` statement is removed as well as the `throw` statement. A `NOP` statement ($TACAI^{L2}$ Line 7) is added because the CFG is not rewritten during the initial transformation, which requires that every basic block contains at least one instruction. It is straight-forward to remove NOPs and update the CFG in a second step.

```
1  RuntimeException e() { return new RuntimeException(); }
2  void p(String s) { System.out.println(s); }
3
4  void m(Serializable serializable) {
5    if(serializable == null) return ;
6    Object o = (Cloneable) serializable;
7    String s = o.toString();
8    StringBuffer sb0 = new StringBuffer();
9    StringBuffer sb1 = sb0.append(s);
10   if(sb0 != sb1)
11     throw e();
12   p(sb0.toString());
13 }
```

Listing 3. Java code used to generate TACAI

3 Evaluation

Next, we evaluate the costs and benefits of our IRs along the following four dimensions:

- **RQ1** How does computing TACAI affect the performance; the time required to compute the IR?
- **RQ2** How does TACAI affect the overall number of three-address code statements?
- **RQ3** In how many cases are we able to provide more precise receiver-type information when compared to the representation offered by the Soot Framework?
- **RQ4** How does it affect the precision of subsequent analyses; in particular call graph algorithms?

Setup. We perform three experiments to answer our research questions. We analyze five programs with `main` method from the XCorpus [6]: jasml, javacc, jext, proguard, sablecc. This is necessary for the call graphs in the third experiment.

All measurements are taken on a Mac Pro with a Xeon E5 with 8 cores@3GHz and a JVM with 24GB of heap space.

Experiment 1. The first experiment aims to answer RQ1 and RQ2 and evaluates how exchanging the abstract interpretation domains affects TACAI's output and its transformation performance. In order to compare the results, we generate Shimple, $TACAI^{L0}$, $TACAI^{L1}$, and $TACAI^{L2}$ (cf. Section 2) for all methods of our evaluation programs.

Table 2 shows the experiment's results. The first three columns show the analyzed project, the number of its classes and methods, respectively. Column four indicates the IR the values in columns five to ten belong to. Those columns present the total number of instructions, the average instruction count per method, its median, and standard deviation. Whereas the call graph's size is irrelevant here, the last column presents the time it takes to generate the IR.

Comparing the runtimes reveals that $TACAI^{L0}$, $TACAI^{L1}$, and $TACAI^{L2}$ are mostly computed significantly faster than Shimple. One exception is *javacc* where $TACAI^{L2}$ took slightly slower than Shimple. The best speedup w.r.t. Shimple of roughly 4.5× is achieved on *proguard*.

We conclude that TACAI's general design is feasible. TACAI can be generated faster than Shimple, even using the most precise configuration $TACAI^{L2}$. Additionally, the overhead to compute $TACAI^{L1}$ compared to $TACAI^{L0}$ is almost negligible. To answer RQ1, computing more precise information takes time. However, when the extra information (e.g. nullness) provided by $TACAI^{L1}$ and $TACAI^{L2}$ are required by an analysis, this time consumption is justifiable.

When we consider the number of instructions (cf. Table 2), its reduction is less than 1 %. Hence, we conclude with regards to RQ2 that the reduction of three-address statements is negligible for our evaluation programs.This is, however, expected because if it would be otherwise, it would indicate dead code [8].

Experiment 2. Here, we compare the type information that is available in Shimple, $TACAI^{L0}$, and $TACAI^{L2}$ in order to answer RQ3. We chose Shimple because it is an SSA-based TAC representation and is thus closer to TACAI than Jimple. WALA's IR does not provide any refined type information over the types available directly in the bytecode. To perform the experiment, we compare each IR's receiver-type information of all potentially polymorphic method invocations.

The comparison across Soot's Shimple and OPAL's TACAI is carried out as follows: First, we generate Shimple for all program methods. Afterward, we traverse each method's Shimple linearly and memorize for each polymorphic invocation its surrounding method, the invoked method's signature, the line numberit occurred in, and its receiver type. Linear

Table 1. Transformed TACAI bytecode from Listing 3 using OPAL's *Level 0*, *Level 1*, and *Level 2* domains. Blue lines mark differences compared to lower levels. Light-blue lines are only syntactic changes.

$TACAI^{L0}$	$TACAI^{L1}$	$TACAI^{L2}$
void m(Serializable) {	void m(Serializable) {	void m(Serializable) {
0: if(p1 ! = null) goto 2	0: if(p1 ! = null) goto 2	0: if(p1 ! = null) goto 2
1: return	1: return	1: return
2: (Cloneable) p1	2: (Cloneable) p1	2: (Cloneable) p1
p1 <: Cloneable	*p1 <: Serializable with Cloneable*	*p1 <: Serializable with Cloneable*
	p1 not null	*p1 not null*
3: lv3 = p1.toString()	3: lv3 = p1.toString()	3: lv3 = p1.toString()
4: lv4 = new StringBuffer	4: lv4 = new StringBuffer	4: lv4 = new StringBuffer
	lv4 not null	*lv4 not null*
5: lv4.<init>()	5: lv4.<init>()	5: lv4.<init>()
6: lv6 = lv4.append(lv3)	6: lv6 = lv4.append(lv3)	6: lv4.append(lv3)
		/* expression value ignored */
7: if(lv4==lv6) goto 10	7: if(lv4==lv6) goto 10	7: ; /* NOP */
8: lv8 = p0.e()	8: lv8 = p0.e()	—
9: throw lv8	9: throw lv8	—
10: lva = lv4.toString()	10: lva = lv4.toString()	8: lv8 = lv4.toString()
11: p0.p(lva)	11: p0.p(lva)	9: p0.p(lv8)
12: return	12: return	10: return
}	}	}

Table 2. Performance results from Experiment 1.

project	#classes	#methods	representation	#instructions	avg.	median	st. dev.	#call edges	runtime
jasml	**50**	**265**	Shimple	-	-	-	-	5 792	7.6s
			$TACAI^{L0}$	14 164	53.5	12	307.5	5 195	3.5s
			$TACAI^{L1}$	14 163	53.5	12	307.5	5 065	3.9s
			$TACAI^{L2}$	14 066	53.4	12	307.5	5 065	6.9s
javacc	**154**	**2151**	Shimple	-	-	-	-	73 884	10.9s
			$TACAI^{L0}$	81 917	38.1	11	150.2	71 515	4.2s
			$TACAI^{L1}$	81 683	38.0	11	150.2	71 003	5.4s
			$TACAI^{L2}$	81 651	38.0	11	150.0	70 985	11.5s
jext	**466**	**2799**	Shimple	-	-	-	-	40 670	19.2s
			$TACAI^{L0}$	73 428	26.2	6	119.8	17 335	4.6s
			$TACAI^{L1}$	73 358	26.2	6	119.8	17 297	5.0s
			$TACAI^{L2}$	73 334	26.2	6	119.7	17 291	6.4s
proguard	**645**	**5237**	Shimple	-	-	-	-	49 260	26.3s
			$TACAI^{L0}$	70 203	13.4	5	140.4	46 218	4.4s
			$TACAI^{L1}$	70 194	13.4	5	140.4	46 096	4.7s
			$TACAI^{L2}$	69 859	13.4	5	140.4	43 535	5.8s
sablecc	**286**	**2274**	Shimple	-	-	-	-	57 021	10.3s
			$TACAI^{L0}$	35 717	15.7	5	50.6	52 076	4.1s
			$TACAI^{L1}$	35 715	15.7	5	50.6	50 939	5.0s
			$TACAI^{L2}$	35 715	15.7	5	50.6	50 939	6.3s

traversal allows us to distinguish multiple invocations within the same line. Then, we generate TACAI in its current configuration and match each call site with those recorded by Soot.

Next, we compare the call site's receiver types to determine if Shimple's type information is more precise than ours or vice versa. If both types are equal, we consider them equally precise if TACAI does not know that its type information is precise, i.e., the exact runtime type is known. In case of precise type information, TACAI is only considered more precise when the precise type has subtypes. When intersection types are inferred, we always consider them to be more precise. However, when TACAI reports union types, we only consider them to be more precise if each type contained in the union is more precise than Shimple's receiver type. Call sites are marked as incomparable when they are not present in either representation.

All results are reported in Table 3 which shows the evaluated project, the compared representations, the project's invocation count, the number of unmatchable call sites, the total number of receiver types that are known to be non-null, the number of invocations with precise receiver-type information, as well as how many call sites the receiver-type information provided by Shimple is equal, better, or worse when compared to TACAI.

Table 3's data shows that we were able to match most call sites across Shimple and TACAI's representation. While comparing both IRs on Proguard, 520 remain unmatched. A closer investigation revealed that Shimple falsely optimizes exception handlers that pertain to JVM-level exceptions (e.g. `ArrayIndexOutOfBoundsException`) which leads to many unmatchable call sites in *Proguard*. Additionally, the call sites that cannot be matched in case of *javacc* are caused by $TACAI^{L2}$'s selective inlining.

When we only consider matchable call sites, we observe that the receiver-type information across Shimple, $TACAI^{L0}$, and $TACAI^{L2}$ are mostly equal. Whereas Shimple never provides more type information than even $TACAI^{L0}$, $TACAI^{L2}$ can maximally improve on *jext* where it has more precise information for 467 receivers. However, the overall number of improvements pertaining to receiver-type information is small. When comparing the availability of nullness information, i.e., the number of cases where we definitively know that a receiver is non-null and no `NullPointerException` can be thrown, between $TACAI^{L0}$ and $TACAI^{L2}$, we observe that non-nullness information is at least available in 11 % of all cases in *sablecc* and up to 40 % of all cases in *jasml*.

Drawing a conclusion to RQ3, we observe that our approach improves little over Shimple w.r.t. receiver-type information. However, $TACAI^{L1}$ and $TACAI^{L2}$ provide additional information useful for static analysis, e.g. nullness.

Experiment 3. Our last experiment evaluates how exchanging abstract domains influences the c all graph construction, answering RQ4. To measure the effect, we construct a class hierarchy analysis (CHA) call graph since it is solely based on the declared types.However, other algorithms (e.g. RTA [12]) may also benefit from more precise type information.

Table 2 provides the call graph's size in the number of edges in the second last column. Whereas we can observe a great reduction of call edges compared to Shimple (up to 58%), the reduction of call edges between TACAI remains minuscule (up to 6%).

Therefore, we observe that the direct impact on call graphs between our IRs for our evaluation set is minor. However, the analyzed programs are rather small in size which lets us assume that the effect on larger programs could increase. More research is necessary to definitely answer RQ4.

4 Related Work

Static analysis tools often work on an intermediate representation (IR) of bytecode which facilitates static analysis. For instance, Soot [13] provides several IRs to operate on: Baf, Jimple, Grimple, and Shimple. However, Jimple and Shimple are the only TAC-based representations. Jimple is generated in 5 steps [14]. At first, a naïve, verbose, and typeless TAC is generated. Step 2 takes the generated TAC and applies several code optimizations, such as constant propagation and dead code elimination. Step 3 splits, step 4 types, and step 5 packs local variables so that they are reused as often as possible. Shimple is produced by converting Jimple into SSA form. In contrast to TACAI, neither Jimple nor Shimple perform all optimizations in one step. Compared to Jimple/Shimple which always provide a type bound, TACAI can provide union and intersection types and derives the information if a specific type is an upper-type bound or a concrete type. Further, TACAI provides a comparable IR when it is configured with its cheapest domain but can be computed faster. Moreover, when advanced domains are configured, TACAI can directly provide additional information, such as def-use information or a variable's nullness.

Demange et al. [5] tackle the problem that an analysis result's correctness strictly depends on the correctness of the performed transformation from the original bytecode to the IR. To mitigate the risk, they provide a formal semantics for an untyped, stack-based Java-like bytecode language, called BC. BC, however, lacks several Java bytecode features (e.g. static fields). Using the defined semantics, they provide a one-pass transformation algorithm that takes BC as input and then generates a TAC-based intermediate representation, called BIR. BIR is proven to preserve the code's semantics. During the transformation, a symbolic stack is used to decompile bytecode into TAC. However, proposed transformation works only for a subset of Java bytecode and does not aim at making the precision configurable.

Table 3. Receiver Type Information of Experiment 2.

project	representation	#inv.	#failed	not null	precise	#equal	#Shimple better	#TACAI better
jasml	Shimple vs $TACAI^{L0}$	2 094	37	0	843	2 057	0	0
	Shimple vs $TACAI^{L2}$	2 094	37	838	1 028	1 987	0	70
javacc	Shimple vs $TACAI^{L0}$	9 883	0	0	4 709	9 878	0	5
	Shimple vs $TACAI^{L2}$	9 722	20	3 551	4 925	9 546	0	164
jext	Shimple vs $TACAI^{L0}$	15 457	2	0	2 803	15 450	0	5
	Shimple vs $TACAI^{L2}$	15 455	2	5 709	3 406	14 986	0	467
proguard	Shimple vs $TACAI^{L0}$	9 961	520	0	3 560	9 439	0	2
	Shimple vs $TACAI^{L2}$	9 959	520	3 694	4 168	9 083	0	356
sablecc	Shimple vs $TACAI^{L0}$	35 717	0	0	4 542	35 716	0	1
	Shimple vs $TACAI^{L2}$	35 715	0	4 143	5 180	35 262	0	453

5 Conclusion

In this paper we presented TACAI, an abstract-interpretation-based intermediate representation with configurable abstract domains. Our intermediate representation directly comes with three preconfigured abstract domains which—when used—result in three-address codes with different levels of precision regarding nullness or available type information.

Our evaluation shows that TACAI is worth researching. It is faster to compute than Soot's Shimple and encodes with $TACAI^{L1}$ and $TACAI^{L2}$ also encodes more information. However, the improvements are minor. In future work, we will further experiment with different domains that can be useful for various static analyses, such as an abstract domain pertaining to the tracking of strings.

Acknowledgments

This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Eric Bodden. 2018. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 85–93.

[2] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 321–333.

[3] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[4] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[5] Delphine Demange, Thomas Jensen, and David Pichardie. 2010. A provably correct stackless intermediate representation for Java bytecode. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–113.

[6] JB Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–An executable Corpus of Java Programs. (2017).

[7] Michael Eichberg and Ben Hermann. 2014. A software product line for static analyses: the OPAL framework. In *SOAP@PLDI*. ACM, 2:1–2:6.

[8] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. 2015. Hidden Truths in Dead Software Paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 474–484.

[9] IBM. [n.d.]. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/. [Online; accessed 19-APRIL-2018].

[10] Steven Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

[11] Yannis Smaragdakis. [n.d.]. DOOP - Framework for Java Pointer and Taint Analysis. https://bitbucket.org/yanniss/doop/. [Online; accessed 23-August-2018].

[12] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices* 35, 10 (2000), 264–280.

[13] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[14] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).