Analysis of Low-Level Code Using Cooperating Decompilers[⋆]

Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula

University of California, Berkeley, California, USA
{bec, matth, necula}@cs.berkeley.edu

Abstract. Analysis or verification of low-level code is useful for minimizing the disconnect between what is verified and what is actually executed and is necessary when source code is unavailable or is, say, intermingled with inline assembly. We present a modular framework for building pipelines of cooperating decompilers that gradually lift the level of the language to something appropriate for source-level tools. Each decompilation stage contains an abstract interpreter that encapsulates its findings about the program by translating the program into a higher-level intermediate language. We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc`, `gcj`, and `coolc` (a compiler for a pedagogical Java-like language) that share several low-level components. Finally, we discuss our experimental results that apply the BLAST model checker for C and the Cqual analyzer to decompiled assembly.

1 Introduction

There is a growing interest in applying software-quality tools to low-level representations of programs, such as intermediate or virtual-machine languages, or even on native machine code. We want to be able to analyze code whose source is either not available (e.g., libraries) or not easily analyzable (e.g., programs written in languages with complex semantics such as C++, or programs that contain inline assembly). This allows us to analyze the code that is actually executed and to ignore possible compilation errors or arbitrary interpretations of underspecified source-language semantics. Many source-level analyses have been ported to low-level code, including type checkers [23, 22, 8], program analyzers [26, 4], model checkers [5], and program verifiers [12, 6]. In our experience, these tools mix the reasoning about high-level notions with the logic for understanding low-level implementation details that are introduced during compilation, such as stack frames, calling conventions, exception implementation, and data layout. We would like to segregate the low-level logic into separate modules to allow for easier sharing between tools and for a cleaner interface with client analyses. To

⋆ This research was supported in part by the National Science Foundation under grants CCF-0524784, CCR-0234689, CNS-0509544, and CCR-0225610; and an NSF Graduate Research Fellowship.

© Springer-Verlag Berlin Heidelberg 2006

better understand this issue, consider developing a type checker similar to the Java bytecode verifier but for assembly language. Such a tool has to reason not only about the Java type system, but also the layout of objects, calling conventions, stack frames, with all the low-level invariants that the compiler intends to preserve. We reported earlier [8] on such a tool where all of this reasoning is done simultaneously by one module. But such situations arise not just for type checking but essentially for all analyses on assembly language.

In this paper we propose an architecture that modularizes the reasoning about low-level details into separate components. Such a separation of low-level logic has previously been done to a certain degree in tools such as CodeSurfer/x86 [4] and Soot [28], which expose to client analyses an API for obtaining information about the low-level aspects of the program. In this paper, we adopt a more radical approach in which the low-level logic is packaged as a *decompiler* whose output is an intermediate language that abstracts the low-level implementation details introduced by the compiler. In essence, we propose that an easy way to reuse source-level analysis tools for low-level code is to decompile the low-level code to a level appropriate for the tool. We make the following contributions:

– We propose a decompilation architecture as a way to apply source-level tools to assembly language programs (Sect. 2). The novel aspect of our proposal is that we use decompilation not only to separate the low-level logic from the source-level client analysis, but also as a way to modularize the low-level logic itself. Decompilation is performed by a series of decompilers connected by intermediate languages. We provide a *cooperation* mechanism in order to deal with certain complexities of decompilation.
– We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc` (for C), `gcj` (for Java), and `coolc` (for Cool [1], a Java-like language used for teaching) that share several low-level components (Sect. 3). We then compare with a monolithic assembly-level analysis.
– We demonstrate that it is possible to apply source-level tools to assembly code using decompilation by applying the BLAST model checker [18] and the Cqual analyzer [17] with our `gcc` decompilation pipeline (Sect. 4).

Note that while ideally we would like to apply analysis tools to machine code binaries, we leave the difficult issue of lifting binaries to assembly to other work (perhaps by using existing tools like IDAPro [19] as in CodeSurfer/x86 [4]).

Challenges. Just like in a compiler, a pipeline architecture improves modularity of the code and allows for easy reuse of modules for different client-analyses. Fig. 1 shows an example of using decompilation modules to process code that has been compiled from C, Java, and Cool. Each stage recovers an abstraction that a corresponding

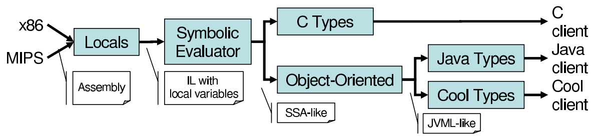


Fig. 1. Cooperating decompilers

compilation stage has concretized. For example, we have a decompiler that decompiles the notion of the run-time stack of activation records into the abstraction of functions with local variables (Locals). The analogy with compilers is very useful but not sufficient. Compilation is in many respects a many-to-one mapping and thus not easily invertible. Many source-level variables are mapped to the same register, many source-level concepts are mapped to the run-time stack, many source-level operations are mapped to a particular low-level instruction kind. We address this issue by providing each decompiler with additional information about the instruction being decompiled. Some information is computed by the decompiler itself using data-flow analysis. For example, the Locals decompiler can keep track of the value of the stack and frame pointer registers relative to function entry.

The real difficulty is that some information must be provided by higher-level modules. For example, the Locals module must identify all calls and determine the number of arguments, but only the object-oriented module (OO) should understand virtual method invocation. There is a serious circularity here. A decompiler needs information from higher-level decompilers to produce the input for the higher-level decompiler. We introduce a couple of mechanisms to address this problem. First, the entire pipeline of decompilers is executed one instruction at a time. That is, we produce decompiled programs simultaneously at all levels. This setup gives each decompiler the opportunity to accumulate data-flow facts that are necessary for decompiling the subsequent instructions and allows the control-flow graph to be refined as the analysis proceeds. When faced with an instruction that can be decompiled in a variety of ways, a decompiler can consult its own data-flow facts and can also query higher-level decompilers for hints based on their accumulated data-flow facts. Thus it is better to think of decompilers not as stages in a pipeline but as cooperating decompilers. The net result is essentially a reduced product analysis [15] on assembly; we explain the benefits of this framework compared to prior approaches based on our previous experiences in Sect. 3 and 5.

2 Cooperating Decompilation Framework

For concreteness, we describe the methodology through an example series of decompiler modules that together are able to perform Java type checking on assembly language. We focus here on the Java pipeline (rather than C), as the desired decompilation is higher-level and thus more challenging to obtain. Consider the example Java program in Fig. 2 and the corresponding assembly code shown in the leftmost column of Fig. 3. In this figure, we use the stack and calling conventions from the x86 architecture where the stack pointer $\mathbf{r}_{sp}$ points to the last used word, parameters are passed on the stack, return values

```
static int length(List x) {
  int n = 0;
  while (x.hasNext()) {
    x = x.next();
    n++;
  }
  return n;
}
```

Fig. 2. A Java method

1	length:	length($\mathbf{t}_x$):	length(α_x):	length(α_x : obj):	length(α_x : *List*):
2	...				
3	$\mathbf{m}[\mathbf{r}_{sp}] := 0$	$\mathbf{t}_n := 0$	$\alpha_n = 0$	$\alpha_n = 0$	$\alpha_n = 0$
4	$\mathrm{L}_{\mathrm{loop}}$:	$\mathrm{L}_{\mathrm{loop}}$:	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_n = \phi(\alpha_n, \alpha'_n)$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_n = \phi(\alpha_n, \alpha'_n)$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_n = \phi(\alpha_n, \alpha'_n)$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$
5	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_{sp}{+}12]$	$\mathbf{r}_1 := \mathbf{t}_x$			
6	jzero $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{exc}}$	jzero $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{exc}}$	if ($\alpha''_x{=}0$) $\mathrm{L}_{\mathrm{exc}}$	if ($\alpha''_x{=}0$) $\mathrm{L}_{\mathrm{exc}}$	if ($\alpha''_x{=}0$) $\mathrm{L}_{\mathrm{exc}}$
7	$\mathbf{r}_2 := \mathbf{m}[\mathbf{r}_1]$	$\mathbf{r}_2 := \mathbf{m}[\mathbf{r}_1]$			
8	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2{+}32]$	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2{+}32]$			
9	$\mathbf{r}_{sp} := \mathbf{r}_{sp} - 4$				
10	$\mathbf{m}[\mathbf{r}_{sp}] := \mathbf{m}[\mathbf{r}_{sp}{+}16]$	$\mathbf{t}_1 := \mathbf{t}_x$			
11	icall $[\mathbf{r}_1]$	$\mathbf{r}_1 :=$ icall $[\mathbf{r}_1](\mathbf{t}_1)$	$\alpha_{rv} =$ icall $[\mathbf{m}[\mathbf{m}[\alpha''_x]{+}32]]$ (α''_x)	$\alpha_{rv} =$ invokevirtual $[\alpha''_x, 32]()$	$\alpha_{rv} =$ α''_x.hasNext()
12	$\mathbf{r}_{sp} := \mathbf{r}_{sp} + 4$				
13	jzero $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{end}}$	jzero $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{end}}$	if ($\alpha_{rv}{=}0$) $\mathrm{L}_{\mathrm{end}}$	if ($\alpha_{rv}{=}0$) $\mathrm{L}_{\mathrm{end}}$	if ($\alpha_{rv}{=}0$) $\mathrm{L}_{\mathrm{end}}$
14	$\mathbf{r}_{sp} := \mathbf{r}_{sp} - 4$				
15	$\mathbf{m}[\mathbf{r}_{sp}] := \mathbf{m}[\mathbf{r}_{sp}{+}16]$	$\mathbf{t}_1 := \mathbf{t}_x$			
16	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2{+}28]$	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2{+}28]$			
17	icall $[\mathbf{r}_1]$	$\mathbf{r}_1 :=$ icall $[\mathbf{r}_1](\mathbf{t}_1)$	$\alpha'_{rv} =$ icall $[\mathbf{m}[\mathbf{m}[\alpha''_x]{+}28]]$ (α''_x)	$\alpha'_{rv} =$ invokevirtual $[\alpha''_x, 28]()$	$\alpha'_{rv} =$ α''_x.next()
18	$\mathbf{r}_{sp} := \mathbf{r}_{sp} + 4$				
19	$\mathbf{m}[\mathbf{r}_{sp}{+}12] := \mathbf{r}_1$	$\mathbf{t}_x := \mathbf{r}_1$	$\alpha'_x = \alpha'_{rv}$	$\alpha'_x = \alpha'_{rv}$	$\alpha'_x = \alpha'_{rv}$
20	incr $\mathbf{m}[\mathbf{r}_{sp}]$	incr $\mathbf{t}_n$	$\alpha'_n = \alpha''_n + 1$	$\alpha'_n = \alpha''_n + 1$	$\alpha'_n = \alpha''_n + 1$
21	jump $\mathrm{L}_{\mathrm{loop}}$	jump $\mathrm{L}_{\mathrm{loop}}$	jump $\mathrm{L}_{\mathrm{loop}}$	jump $\mathrm{L}_{\mathrm{loop}}$	jump $\mathrm{L}_{\mathrm{loop}}$
22	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:
23	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_{sp}]$	$\mathbf{r}_1 := \mathbf{t}_n$			
24	...				
25	return	return $\mathbf{r}_1$	return α''_n	return α''_n	return α''_n
	Assembly	Locals IL	SymEval IL	OO IL	Java IL

Fig. 3. Assembly code for the program in Fig. 2 and the output of successive decompilers. The function's prologue and epilogue have been elided. Jumping to $\mathrm{L}_{\mathrm{exc}}$ will trigger a Java `NullPointerException`.

are passed in $\mathbf{r}_1$, and $\mathbf{r}_2$ is a callee-save register. Typically, a virtual method dispatch is translated to several lines of assembly (e.g., lines 6–11): a null-check on the receiver object, looking up the dispatch table, and then the method in the dispatch table, passing the receiver object and any other arguments, and finally an indirect jump-and-link (`icall`). To ensure that the `icall` is a correct compilation of a virtual method dispatch, dependencies between assembly instructions must be carefully tracked, such as the requirement that the argument passed as the self pointer is the same as the object from which the dispatch table is obtained (cf., [8]). These difficulties are only exacerbated with instruction reordering and other optimizations. For example, consider the assembly code for the method dispatch to `x.next`() (lines 14–17). Variable `x` is kept in a stack slot ($\mathbf{m}[\mathbf{r}_{sp}{+}16]$ at line 15). A small bit of optimization has eliminated the null-check and the re-fetching of the dispatch table of `x`, as a null-check was done on line 6 and the dispatch table was kept in a callee-save register $\mathbf{r}_2$, so clearly some analysis is necessary to decompile it into a method call.

The rest of Fig. 3 shows how this assembly code is decompiled by our system. Observe how high-level constructs are recovered incrementally to obtain essentially Java with unstructured control-flow (shown in the rightmost column). Note that our goal is not to necessarily recover the same source code but simply code that is semantically equivalent and amenable to further analysis. To summarize the decompilation steps, the Locals module decompiles stack and calling conventions to provide the abstraction of functions with local variables. The SymEval decompiler performs symbolic evaluation to accumulate and normalize larger expressions to present the program in a source-like SSA form. Object-oriented features, like virtual method dispatch, are identified by the OO module, which must understand implementation details like object layout and dispatch tables. Finally, JavaTypes can do a straightforward type analysis (because its input is already fairly high-level) to recover the Java-like representation.

As can be seen in Fig. 3, one key element of analyzing assembly code is decoding the run-time stack. An assembly analyzer must be able to identify function calls and returns, recognize memory operations as either stack accesses or heap accesses, and must ensure that stack-overflow and calling conventions are handled appropriately. This handling ought to be done in a separate module both because it is not specific to the desired analysis and also to avoid such low-level concerns when thinking about the analysis algorithm (e.g., Java type-checking). In our example decompiler pipeline (Fig. 1), the Locals decompiler handles all of these low-level aspects. On line 17, the Locals decompiler determines that this instruction is a function call with one argument (for now, we elide the details how this is done, see the Bidirectional Communication subsection and Fig. 4). It interprets the calling convention to decompile the assembly-level jump-and-link instruction to a function call instruction with one argument that places its return value in $\mathbf{r}_{rv}$. Also, observe that Locals decompiles reads of and writes to stack slots that are used as local variables into uses of *temporaries* (e.g., $\mathbf{t}_x$) (lines 3, 5, 10, 15, 19, 20, 23). To do these decompilations, the Locals decompiler needs to perform analysis to track, for example, pointers into the stack. For instance, Locals needs this information to identify the reads on both lines 5 and 10 as reading the same stack slot $\mathbf{t}_x$. Section 3 gives more details about how these decompilers are implemented.

Decompiler Interface. Program analyses are almost always necessary to establish the prerequisites for sound decompilations. We build on the traditional notions of data-flow analysis and abstract interpretation [14]. Standard ways to combine abstract interpreters typically rely on all interpreters working on the same language. Instead, we propose here an approach in which the communication mechanism consists of successive decompilations. Concretely, a decompiler must define a type of abstract states `abs`

```
type abs
  val step : curr × instr_in → instr_out × (succ list)
  val ⊑ : abs × abs → bool
  val ∇ : abs × abs → abs
```

and implement a flow function (i.e., abstract transition relation) `step` with the type signature given above for some input language $\texttt{instr}_{in}$ and some output language $\texttt{instr}_{out}$. The input type `curr` represents the abstract state at the given instruction, and `succ` is an abstract successor state at a particular program location. For simplicity in presentation, we say a decompiler translates one input instruction to one output instruction. Our implementation extends this to allow one-to-many or many-to-one translations. As part of the framework, we provide a standard top-level fixed-point engine that ensures the exploration of all reachable instructions. To implement this fixed-point engine, we require the signature include the standard partial ordering $\sqsubseteq$ and widening $\triangledown$ operators [14] for abstract states.

For simple examples where the necessary communication is unidirectional (that is, from lower-level decompilers to higher-level decompilers via the decompiled instructions), an exceedingly simple composition strategy suffices where we run each decompiler completely to fixed point gathering the entire decompiled program before running the next one (i.e., a strict pipeline architecture). This architecture does not require a product abstract domain and would be more efficient than one. Unfortunately, as we have alluded to earlier, unidirectional communication is insufficient: lower-level decompilers depend on the analyses of higher-level decompilers to perform their decompilations. We give examples of such situations and describe how to resolve this issue in the following subsection.

Bidirectional Communication. In this subsection, we motivate two complimentary mechanisms for communicating information from higher-level decompilers to lower-level ones. In theory, either mechanism is sufficient for all high-to-low communication but at the cost of efficiency or naturalness. As soon as we consider high-to-low communication, clearly the strict pipeline architecture described above is insufficient: higher-level decompilers must start before lower-level decompilers complete. To address this issue, we run the entire pipeline of decompilers one instruction at a time, which allows higher-level decompilers to analyze the preceding instructions before lower-level decompilers produce subsequent instructions. For this purpose, we provide a product decompiler whose abstract state is the product of the abstract states of the decompilers, but in order to generate its successors, it must string together calls to `step` on the decompilers in the appropriate order and then collect together the abstract states of the decompilers.

Queries. Consider again the dynamic dispatch on line 17 of Fig. 3. In order for the Locals module to (soundly) abstract stack and calling conventions into functions with local variables, it must enforce basic invariants, such as a function can only modify stack slots (used as temporaries) in its own activation record (i.e., stack frame). To determine the extent of the callee's activation record, the Locals module needs to know, among other things, the number of arguments of the called function, but only the higher-level decompiler that knows about the class hierarchy (JavaTypes) can determine the calling convention of the methods that $\mathbf{r}_1$ can possibly point to. We resolve this issue by allowing lower-level decompilers to query higher-level decompilers for hints. In this case, Locals asks:

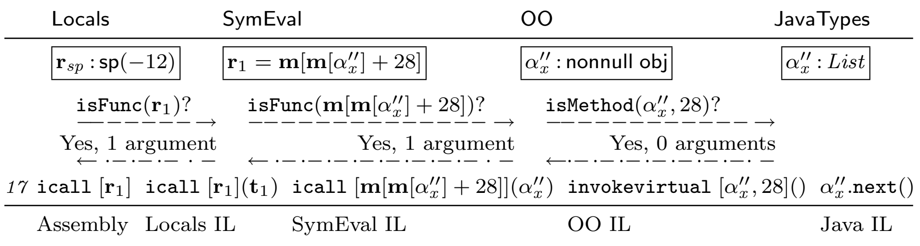

Locals
SymEval
OO
JavaTypes
$\boxed{\mathbf{r}_{sp} : \mathsf{sp}(-12)}$
$\boxed{\mathbf{r}_1 = \mathbf{m}[\mathbf{m}[\alpha''_x] + 28]}$
$\boxed{\alpha''_x : \mathsf{nonnull\ obj}}$
$\boxed{\alpha''_x : \mathit{List}}$
$\mathtt{isFunc}(\mathbf{r}_1)?$
$\mathtt{isFunc}(\mathbf{m}[\mathbf{m}[\alpha''_x] + 28])?$
$\mathtt{isMethod}(\alpha''_x, 28)?$
Yes, 1 argument
Yes, 1 argument
Yes, 0 arguments
$\mathit{17}$ $\mathtt{icall}\ [\mathbf{r}_1]$
$\mathtt{icall}\ [\mathbf{r}_1](\mathbf{t}_1)$
$\mathtt{icall}\ [\mathbf{m}[\mathbf{m}[\alpha''_x] + 28]](\alpha''_x)$
$\mathtt{invokevirtual}\ [\alpha''_x, 28]()$
$\alpha''_x.\mathtt{next}()$
Assembly
Locals IL
SymEval IL
OO IL
Java IL

Fig. 4. Queries to resolve the dynamic dispatch from line 17 of Fig. 3

"Should `icall` $[\mathbf{r}_1]$ be treated as a standard function call; if so, how many arguments does it take?". If some higher-level decompiler knows the answer, then it can translate the assembly-level jump-and-link (`icall` $[\mathbf{r}_1]$) to a higher-level call with arguments and a return register and appropriately take into account its possible interprocedural effects.

In Fig. 4, we show this query process in further detail, eliding the return values. Precisely how these decompilers work is not particularly relevant here (see details in Sect. 3). Focus on the original query `isFunc`$(\mathbf{r}_1)$ from Locals. To obtain an answer, the query gets decompiled into appropriate variants on the way up to JavaTypes. The answer is then translated on the way down. For the OO module the method has no arguments, but at the lower-level the implicit `this` argument becomes explicit. For JavaTypes to answer the query, it must know the type of the receiver object, which it gets from its abstract state. The abstract states of the intermediate decompilers are necessary in order to translate queries so that JavaTypes can answer them. We show portions of each decompiler's abstract state in the boxes above the queries; for example, Locals must track the current value of the stack pointer register $\mathbf{r}_{sp}$ (we write $\mathsf{sp}(n)$ for a stack pointer that is equal to $\mathbf{r}_{sp}$ on function entry plus n). By also tracking return addresses, this same query also allows Locals to decompile calls that are implemented in assembly as (indirect) `jump`s (e.g., tail calls). This canonicalization then enables higher-level decompilers to treat all calls uniformly.

$\boxed{\textbf{type}\ \mathtt{curr} = \mathtt{hints}_{out} \times \mathtt{abs}}$

Adjacent decompilers agree upon the queries that can be made by defining a type `hints` in their shared intermediate language. An object of type $\mathtt{hints}_{out}$ provides information about the current abstract states of higher-level decompilers, usually in the form of one or more callback functions like `isFunc`. Such an object is provided as an input to the `step` function of each decompiler (as part of `curr`); This architecture with decompilations and callbacks works quite nicely, as long as the decompilers agree on the number of successors and their program locations.

Decompiling Control-Flow. Obtaining a reasonable control-flow graph on which to perform analysis is a well-known problem when dealing with assembly code and is often a source of unsoundness, particularly when handling indirect control-flow. For example, switch tables, function calls, function returns, exception raises may all be implemented as indirect jumps (`ijump`) in assembly. We approach this problem by integrating the control-flow determination with the decompilation; that is, we make no *a priori* guesses on where an indirect jump goes and rely on the decompiler modules to resolve them to a set of concrete program points. In general, there are two cases where the decompilers may not be able to agree on the same successors: lower-level decompilers *don't know the successors* or higher-level ones have *additional successors*. Sometimes a low-level decompiler does not know the possible concrete successors. For example, if the Locals decompiler cannot resolve an indirect jump, it will produce an *indirect* successor indicating it does not know where the indirect jump will go. However, a higher-level decompiler may be able to refine the indirect successor to a set of concrete successors (that, for soundness, must cover where the indirect jump may actually go). It is then an error if any indirect successors remain unresolved after the entire pipeline. A decompiler may also need to introduce additional successors not known to lower-level modules. In both examples, a high-level decompiler augments the set of successors with respect to those of the low-level decompilers. The problem is that we do not have abstract states for the low-level decompilers at the newly introduced successors. This, in turn, means that it will be impossible to continue the decompilation at one of these successors.

To illustrate the latter situation, consider a static method call `C.m()` inside the `try` of a `try-catch` block and its compilation to assembly (shown to the right). We would like to make use of the run-time stack analysis and expression normalization performed by Locals and SymEval in decompiling exceptions, so the decompiler that handles exceptions should be placed somewhere after them in the pipeline. However, the Locals decompiler, and several decompilers after it, produce one successor abstract state after the call to `C.m()` (line 2). In order to soundly analyze a possible `throw` in `C.m()`, the decompiler that handles exceptions must add one more successor at the method call for the `catch` block at L_{catch}. The challenge is to generate appropriate low-level abstract states for the successor at L_{catch}. For example, the exceptions decompiler might want to direct all other decompilers to transform their abstract states before the static method call and produce an abstract state for L_{catch} from it by clobbering certain registers and portions of memory.

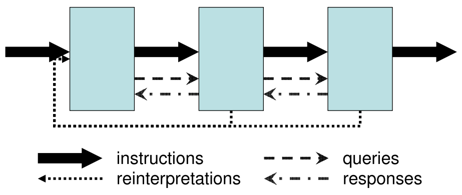
```
1    ...
2    call C.m
3    ...
4    jump L_exit
5  L_catch :
6    ...
7  L_exit :
8    ...
```

The mechanism we propose is based on the observation that we already have a pipeline of decompilers that is able to transform the abstract states at all levels when given a sequence of machine instructions. To take advantage of this we require a decompiler to provide, for each newly introduced successor, a list of machine instructions that will be "run" through the decompilation pipeline (using `step`) to produce the missing lower-level abstract states. To achieve this, the `succ` type (used in the return of `step`) carries an optional list of *machine* instructions (of type $\texttt{instr}_C$). As a side-condition, the concrete machine instructions returned by `step` should not include control-flow instructions (e.g., `jump`). We also extend the concrete machine instruction set

```
type succ = loc × (abs × ((instr_C list) option))
```

with instructions for abstracting effects; for example, there is a way to express that register $\mathbf{r}_x$ gets modified arbitrarily (`havoc` $\mathbf{r}_x$).

Both queries and these *reinterpretations* introduce a channel of communication from higher-level decompilers to lower-level ones, but they serve complimentary purposes. For one, reinterpretations are initiated by high-level decompilers, while queries are initiated by low-level decompilers. We want to use queries when we want the question to be decompiled, while we prefer to communicate through reinterpretations when we want the answers to be decompiled. The diagram above summarizes these points. In the extended version [9], we give the product decompiler that ties together the pipeline (with queries and reinterpretations), which further clarifies how the decompiler modules interact to advance simultaneously.



Soundness of Decompiler Pipelines. One of the main advantages of the modular architecture we describe in this paper is that we can modularize the soundness argument itself. This modularization increases the trustworthiness of the program analysis and is a first step towards generating machine-checkable proofs of soundness, in the style of Foundational Proof-Carrying Code [3].

Since we build on the framework of abstract interpretation, the proof obligations for demonstrating the soundness of a decompiler are fairly standard local criteria, which we sketch here. Soundness of a decompiler module is shown with respect to the semantics of its *input and output* languages given by concrete transition relations. In particular, leaving the program implicit, we write $I_{\mathcal{L}} \mathbin{⨟} l \rightsquigarrow_{\mathcal{L}} l'@\ell$ for the one-step transition relation of the input (lower-level) machine, which says that on instruction $I_{\mathcal{L}}$ and pre-state l, the post-state is l' at program location ℓ (similarly for the output machine $\mathcal{H}$). As usual, we can specify whatever safety policy of interest by disallowing transitions that would violate the policy (i.e., modeling errors as "getting stuck"). Also, we need to define a *soundness relation* $l \precsim a$ between concrete states for the input machine and abstract states, as well as a *simulation relation* $l \sim h$ between concrete states of the input and output machines.

Note that for a given assembly program, we use the same locations for all decompilations since we consider one-to-one decompilations for presentation purposes (otherwise, we would consider a correspondence between locations at different levels). Let $\mathcal{L}_0$ and $\mathcal{H}_0$ denote the initial machine states (as a mapping from starting locations to states) such that they have the same starting locations each with compatible states (i.e., $\mathrm{dom}(\mathcal{L}_0) = \mathrm{dom}(\mathcal{H}_0)$ and $\mathcal{L}_0(\ell) \sim \mathcal{H}_0(\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{L}_0)$). Now consider running the decompiler pipeline to completion (i.e., to fixed point) and let $\mathcal{A}_{\text{INV}}$ be the mapping from locations to abstract states at fixed point. Note that $\mathcal{A}_{\text{INV}}$ must contain initial abstract states compatible with the concrete states in $\mathcal{L}_0$ (i.e., $\mathrm{dom}(\mathcal{L}_0) \subseteq \mathrm{dom}(\mathcal{A}_{\text{INV}})$ and $\mathcal{L}_0(\ell) \precsim \mathcal{A}_{\text{INV}}(\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{L}_0)$).

We can now state the local soundness properties for a decompiler module's `step`. A decompiler's `step` need only give sound results when the query object it receives as input yields answers that are sound approximations of the machine state, which we write as $h \precapprox q$ (and which would be defined and shown separately).

Property 1 (Progress). If $l \sim h$, $l \precsim a$, $h \precapprox q$, $\texttt{step}((q, a), I_{\mathcal{L}}) = (I_{\mathcal{H}}, A')$ and $I_{\mathcal{H}} \mathbin{⨾} h \rightsquigarrow_{\mathcal{H}} h'@\ell$, then $I_{\mathcal{L}} \mathbin{⨾} l \rightsquigarrow_{\mathcal{L}} l'@\ell$ (for some l').

Progress says that whenever the decompiler can make a step *and* whenever the output machine is not stuck, then the input machine is also not stuck. That is, a decompiler residuates soundness obligations to higher-level decompilers through its output instruction. Thus far, we have not discussed the semantics of the intermediate languages very precisely, but here is where it becomes important. For example, for stack slots to be soundly translated to temporaries by the Locals decompiler, the semantics of the memory write instruction in Locals IL is not the same as a memory write in the assembly in that it must disallow updating such stack regions. In essence, the guarantees provided by and the expectations of a decompiler module for higher-level ones are encoded in the instructions it outputs. If a decompiler module fails to perform sufficient checks for its decompilations, then the proof of this property will fail.

To implement a verifier that enforces a particular safety policy using a decompiler pipeline, we need to have a module at the end that does not output higher-level instructions to close the process (i.e., capping the end). Such a module can be particularly simple; for example, we could have a module that simply checks syntactically that all the “possibly unsafe” instructions have been decompiled away (e.g., for memory safety, all memory read instructions have been decompiled into various safe read instructions).

Property 2 (Preservation). If $l \sim h$, $l \precsim a$, $h \precapprox q$ and $\texttt{step}((q, a), I_{\mathcal{L}}) = (I_{\mathcal{H}}, A')$, then for every l' such that $I_{\mathcal{L}} \mathbin{⨾} l \rightsquigarrow_{\mathcal{L}} l'@\ell$, there exists h', a' such that $I_{\mathcal{H}} \mathbin{⨾} h \rightsquigarrow_{\mathcal{H}} h'@\ell$ where $l' \sim h'$ and $a' = \mathcal{A}_{\text{INV}}(\ell)$ where $l' \precsim a'$.

Preservation guarantees that for every transition made by the input machine, the output machine simulates it and the concrete successor state matches one of the abstract successors computed by $\texttt{step}$ (in $\mathcal{A}_{\text{INV}}$).

3 Decompiler Examples

In this section, we describe a few decompilers from Fig. 1. For each decompiler, we give the instructions of the output language, the lattice of abstract values, and a description of the decompilation function $\texttt{step}$. We use the simplified notation $\texttt{step}(a_{curr}, I_{in}) = (I_{out}, a_{succ})$ to say that in the abstract state a_{curr} the instruction I_{in} is decompiled to I_{out} and yields a successor state a_{succ}. We write $a_{succ}@\ell$ to indicate the location of the successor, but we elide the location in the common case when it is “fall-through”. A missing successor state a_{succ} means that the current analysis path ends. We leave the query object implicit, using q to stand for it when necessary. Since each decompiler has similar structure, we use subscripts with names of decompilers or languages when necessary to clarify to which module something belongs.

Decompiling Calls and Locals. The Locals module deals with stack conventions and introduces the notion of statically-scoped local variables. The two major changes from assembly instructions (I_C) are that call and return instructions have actual arguments. The abstract state includes a mapping Γ from variables x to abstract values τ, along with two additional integers, n_{lo} and n_{hi}, that delimit the current activation record (i.e., the extent of the known valid stack addresses for this function) with respect to the value of the stack pointer on entry. The variables mapped by the abstract state include all machine registers and variables $\mathbf{t}_n$ that correspond to stack slots (with the subscript indicating the stack offset of the slot in question). We need only track a few abstract values τ: the value of stack pointers $\mathsf{sp}(n)$, the return address for the function ra, code addresses for function return addresses $\&\ell$, and the value of callee-save registers on function entry $\mathsf{cs}(r)$. These values form a flat lattice, with the usual ordering.

$$\begin{array}{lrl} \texttt{instr} & I_L ::= & I_C \mid x := \texttt{call}\ \ell(e_1, ..., e_n) \\ & & \mid x := \texttt{icall}\ [e](e_1, ..., e_n) \\ & & \mid \texttt{return}\ e \\ \text{abs values} & \tau ::= & \top \mid n \mid \mathsf{sp}(n) \mid \mathsf{ra} \mid \&\ell \mid \mathsf{cs}(r) \end{array}$$

Many of the cases for the `step` function propagate the input instruction unchanged and update the abstract state. We show here the definition of `step` for the decompilation of a stack memory read to a move from a variable. For simplicity, we assume here that all stack slots are used for locals. This setup can be extended to allow higher-level decompilers to indicate (through some high-to-low communication) which portions of the stack frame it wants to handle separately. We write $\Gamma \vdash e : \tau$ to say that in the abstract state $\langle \Gamma; n_{lo}; n_{hi} \rangle$, the expression e has abstract value τ. For verifying memory safety, a key observation is that Locals proves once and for all that such a read is to a valid memory address; by decompiling to a move instruction, no higher-level decompiler needs to do this reasoning. The analogous translation for stack writes appears on, for example, line 19 in Fig. 3.

$$\frac{\Gamma \vdash e : \mathsf{sp}(n) \quad n_{lo} \le n \le n_{hi} \quad n \equiv 0 \pmod 4}{\texttt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, r := \mathbf{m}[e]) \ = \ (r := \mathbf{t}_n, \langle \Gamma[r \mapsto \Gamma(\mathbf{t}_n)]; n_{lo}; n_{hi} \rangle)}$$

The following rule gives the translation of function calls:

$$\frac{\Gamma(x_{ra}) = \&\ell \quad \Gamma(\mathbf{r}_{sp}) = \mathsf{sp}(n) \quad n \equiv 0 \pmod 4 \quad q.\texttt{isFunc}(e) = k \quad \Gamma' = \mathit{scramble}(\Gamma, n, k)}{\texttt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, \texttt{icall}\ [e]) \ = \ (x_{rv} := \texttt{icall}\ [e](x_1, ..., x_k), \langle \Gamma'[\mathbf{r}_{sp} \mapsto \mathsf{sp}(n{+}4)]; n_{lo}; n_{hi} \rangle @\ell)}$$

It checks that the return address is set, $\mathbf{r}_{sp}$ contains a word-aligned stack pointer, and e is the address of a function according to the query. Based on the calling convention and number of arguments, it constructs the call with arguments and the return register. The successor state Γ' is obtained first by clearing any non-callee-save registers and temporaries corresponding to stack slots in the callee's activation record, which is determined by *scramble* using the calling convention, n, and k. Then, $\mathbf{r}_{sp}$ is updated, shown here according to the x86 calling convention where the callee pops the return address. In the implementation, we parameterize by a description of the calling convention. Further details, including the verification of stack overflow checking, is given in the extended version [9].

Symbolic Evaluator. The SymEval ($\mathcal{E}$) module does the following analysis and transformations for higher-level decompilers to resolve some particularly pervasive problems when analyzing assembly code.

1. *Simplified and Normalized Expressions.* High-level operations get compiled into long sequences of assembly instructions with intermediate values exposed (as exemplified in Fig. 3). To analyze one instruction at a time, we need to assign types to all intermediate expressions, but this undertaking quickly becomes unwieldy. Additionally, arithmetic equivalences are used extensively by compilers (particularly in optimized code). We want to accumulate larger expression trees and perform arithmetic simplification and normalization before assigning types. Observe how SymEval does this work in the example decompilation of line 17 in Fig. 4.
2. *Static Single Assignment (SSA).* In contrast to source-level variables, flow-sensitivity is generally required to analyze registers because registers are reused for unrelated purposes. To have a set variables suitable for source-level analyses, the symbolic evaluator yields an SSA-like program representation.
3. *Global Value Numbering (GVN).* The same variable may also be placed in multiple locations (yielding an equality on those locations). For example, to check that a reference stored on the stack is non-null, a compiler must emit code that first loads it into a register. On the non-null path, an assembly-level analysis needs to know that the contents of both the register and the stack slot is non-null. So that higher-level decompilers do not have to deal with such low-level details, the symbolic evaluator presents a single *symbolic value* α that abstracts some unknown value but is stored in both the register and the stack slot (implicitly conveying the equality). Combined with the above, the symbolic evaluator can be viewed as implementing an extended form of GVN [2]. Further details are given in the extended version [9].

Decompiling Object-Oriented Features. The OO decompiler ($\mathcal{O}$) recognizes compilations of class-based object-oriented languages, such as Java. The output instruction language for the OO decompiler includes the instructions from the symbolic evaluator, except it is extended for virtual method dispatch, field reads, and field writes. Almost all of the heavy lifting has been done by the symbolic evaluator, so OO is quite simple. The abstract values that we need to track are straightforward: a type for object references, which may be qualified as non-null or possibly null.

$$\begin{array}{lll} \texttt{instr} & I_O ::= I_{\mathcal{E}} \mid \alpha = \texttt{putfield}\ [e, n] \\ & \qquad\quad \mid \alpha = \texttt{invokevirtual}\ [e_0, n](e_1, ..., e_n) \\ \texttt{expr} & e_O ::= e_{\mathcal{E}} \mid \texttt{getfield}\ [e, n] \end{array}$$

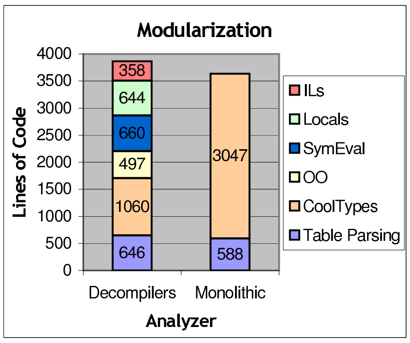
The decompilation of virtual method dispatch (as on line 17 in Fig. 4) is as follows:

$$\frac{\Gamma(\beta) = \textsf{nonnull obj} \quad \Gamma \vdash e_1 : \tau_1 \ \cdots\ \Gamma \vdash e_m : \tau_m \quad q.\texttt{isMethod}(\beta, n) = \tau_1 \times \cdots \times \tau_m \to \tau}{\begin{array}{l}\texttt{step}(\Gamma, \alpha = \texttt{icall}\ [\mathbf{m}[\mathbf{m}[\beta] + n]](\beta, e_1, ..., e_m)) \\ \qquad = (\alpha = \texttt{invokevirtual}\ [\beta, n](e_1, ..., e_m), \Gamma[\alpha \mapsto \tau])\end{array}}$$

It checks that the object reference is non-null and that the dispatch table is obtained from the same object as the object being passed as the receiver object. Observe that since the abstract state is independent of the register and memory state, the successor abstract state is particularly easy to derive. One additional bit of interesting work is that it must recognize null-checks and strengthen a possibly-null object to a non-null one. Because of the symbolic evaluator, OO simply updates the type of a symbolic value α and need not worry about the equivalences between all the registers or temporaries that contain α.

Implementation and Experience. We have implemented and tested the above decompiler modules in multiple decompiler pipelines, including three main ones for assembly generated from Java programs by `gcj`, C programs by `gcc`, and Cool programs by `coolc`. All decompiler pipelines start from a very simple untyped RISC-like assembly language to minimize architecture dependence. We have parsers for x86 and MIPS that translate to this generic assembly. The Locals module is parameterized by the calling convention, so we can easily handle several different calling conventions (e.g., standard x86, standard MIPS, or the non-standard one used by `coolc`). In these pipelines, we use communication in three main ways: queries for identifying function or method calls (as in Fig. 4), queries for pointer types, and reinterpretations for exceptional successors (as in Decompiling Control-Flow of Sect. 2). The responses for the `isFunc` and `isMethod` queries contain a bit more information than as shown in Fig. 4, such as the calling convention for the callee and between JavaTypes / CoolTypes and OO, the types of the parameters and the return value (i.e., whether they are object references). The OO decompiler also queries JavaTypes / CoolTypes to determine certain pointer types that may require consulting the class table, such as whether a read field is an object reference.

Each of the decompiler modules described above is actually quite small (at most ~600 lines of OCaml). Furthermore, each module is approximately the same size providing some evidence for a good division of labor. The overhead (i.e., the definition of the intermediate languages and associated utility functions) seems reasonable, as each language only required 100−150 lines of OCaml. The entire `coolc` pipeline (including the Cool type analysis but not the framework code) is 3,865 lines compared to 3,635 lines for a monolithic assembly-level analyzer from our previous work [8], which uses the classic reduced product approach (as shown visually above). Cool is a fairly realistic subset of Java, including features such as exceptions, so the CoolTypes module includes the handling of exceptions as described in Decompiling Control-Flow of Sect. 2. The additional code is essentially in the definition of the intermediate languages, so what we conclude is that our pipeline approach does give us a modular and easier to maintain design without imposing an unreasonable code size penalty



with respect to the monolithic version. Additionally, note that 2,159 and 1,515 of the 3,865 lines of the `coolc` decompiler pipeline are reused as-is in the `gcj` and `gcc` pipelines, respectively.

Comparing the implementation experience with our previous assembly-level analyzer, we found that the separation of concerns imposed by this framework made it much easier to reason about and implement such assembly-level analyses. For example, because of the decompilations, Cool/Java type inference is no longer intermingled with the analysis of compiler-specific run-time structures. With this framework, we also obtained comparable stability in a much shorter amount of time. Many of the bugs in the implementation described in our prior work [8] were caused by subtle interactions in the somewhat ad-hoc modularization there, which simply did not materialize here. Concretely, after testing our `coolc` decompiler pipeline on a small suite of regression tests developed with the previous monolithic version, we ran both the decompiler pipeline and the previous monolithic versions on the set of 10,339 test cases generated from Cool compilers developed by students in the Spring 2002, Spring 2003, and Spring 2004 offerings of the compilers course at UC Berkeley (on which we previously reported [8]). Of the 10,339 test cases, they disagreed in 182 instances, which were then examined manually to classify them as either soundness bugs or incompletenesses in either the decompiler or monolithic versions. We found 1 incompleteness in the decompiler version with respect to the monolithic version that was easily fixed (some identification of dead code based on knowing that a pointer is non-null), and we found 0 soundness bugs in the decompiler version. At the same time, we found 5 incompletenesses in the monolithic version; in 2 cases, it appears the SymEval module was the difference. Surprisingly, we found 3 soundness bugs in the monolithic version, which has been used extensively by several classes. We expected to find bugs in the decompiler version to flush out, but in the end, we actually found more bugs in the more well-tested monolithic version. At least 1 soundness bug and 1 incompleteness in the monolithic version were due to mishandling of calls to run-time functions. There seem to be two reasons why the decompiler version does not exhibit these bugs: the updating of effects after a call is implemented in several places in the monolithic version (because of special cases for run-time functions), while in the decompiler version, the Locals decompiler identifies all calls, so they can be treated uniformly in all later modules; and the SSA-like representation produced by SymEval decompiler greatly simplifies the handling of interprocedural effects in higher-level modules.

As another example of the utility of this approach, after the implementation for the class table parser was complete (which are already generated by `gcj` to support reflection), one of the authors was able to implement a basic Java type inference module in 3–4 hours and $\sim$500 lines of code (without the handling of interfaces and exceptions).

4 Case Studies

To explore the feasibility of applying existing source-level tools to assembly code, we have used BLAST [18] and Cqual [17] on decompilations produced by our `gcc` pipeline. To interface with these tools, we have a module that emits C from SymEval IL. SymEval IL is essentially C, as register reuse with unrelated types have been eliminated by SSA and expression trees have been recovered. However, while a straightforward translation from SymEval IL produces a valid C program that can be (re)compiled and executed, the typing is often too weak for source-level analysis tools. To avoid this issue for these experiments, we use debugging information to recover types. When debugging information is not available, we might be able to obtain typing information using a decompiler module that implements a type reconstruction algorithm such as Mycroft's [24].

We have taken the benchmarks shown in the table, compiled them to x86 (unoptimized), and decompiled them back to C before feeding the decompilations to the source-level tools (B for BLAST and Q for Cqual). In all cases, we checked that the tools could verify the presence (or absence) of bugs just as they had for the original C program. In the table, we show our decompilation times and the verification times of both the original and decompiled programs on a 1.7GHz Pentium 4 with 1GB RAM. The BLAST cases `qpmouse.c` and `tlan.c` are previously reported Linux device drivers for which BLAST checks that `lock` and `unlock` are used correctly [18]. For `gamma_dma.c`, a file from version 2.4.23 of the Linux kernel, Cqual is able to find in the decompiled program a previously reported bug involving the unsafe dereference of a user-mode pointer [20]. Both Cqual and BLAST require interprocedural analyses and some C type information to check their respective properties. We have also repeated some of these experiments with optimized code. With `qpmouse`, we were able to use all the `-O2` optimizations in `gcc` 3.4.4, such as instruction scheduling, except `-fmerge-constants`, which yields code that reads a byte directly from the middle of a word-sized field, and `-foptimize-sibling-calls`, which introduces tail calls. The latter problem we could probably handle with an improved Locals module, but the former is more difficult due to limitations with using the debugging information for recovering C types. In particular, it is challenging to map complicated pointer offsets back to C `struct` accesses. Similarly, it is sometimes difficult to insert casts that do not confuse client analyses based only on the debugging information because it does not always tell us where casts are performed. Finally, we do not yet handle all assembly instructions, particularly kernel instructions.

Test Case		Code Size C (loc)	Code Size x86 (loc)	Decomp. (sec)	Verification Orig. (sec)	Verification Decomp. (sec)
`qpmouse.c`	(B)	7994	1851	0.74	0.34	1.26
`tlan.c`	(B)	10909	10734	8.16	41.20	94.30
`gamma_dma.c`	(Q)	11239	5235	2.44	0.97	1.05

5 Related Work

In abstract interpretation, the problem of combining abstract domains has also been considered by many. Cousot and Cousot [15] define the notion of a *reduced product*, which gives a "gold standard" for precise combinations of abstract domains. Unfortunately, obtaining a reduced product implementation is not automatic; they generally require manual definitions of reduction operators, which depend on the specifics of the domains being combined (e.g., [11]). Roughly speaking, we propose a framework for building reduced products based on decompilation, which is particular amiable for modularizing the analysis of assembly code. Cortesi *et al.* [13] describe a framework (called an *open product*) that takes queries as the central (and only) means of communication. They allow arbitrary queries between any pair of domains, whereas our queries are more structured through decompilation. With this structure we impose, modules need only agree upon a communication interface with its neighbors. Combining program analyses for compiler optimization is also a well-known and well-studied problem. Lerner *et al.* [21] propose modular combinations of compiler optimizations also by integrating analysis with program transformation, which then serve as the primary channel of communication between analyses. We, however, use transformation for abstraction rather than optimization. For this reason, we use layers of intermediate languages instead of one common language, which is especially useful to allow residuation of soundness obligations.

Practically all analysis frameworks, particularly for low-level code, perform some decompilation or canonicalization for client analyses. For example, CodeSurfer/x86 [4] seeks to provide a higher-level intermediate representation for analyzing x86 machine code. At the core of CodeSurfer/x86 is a nice combined integer and pointer analysis (*value set analysis*) for abstract locations. The motivation for this analysis is similar to that for the Locals module, except we prefer to handle the heap separately in language-specific ways. Their overall approach is a bit different from ours in that they try to decompile without the assistance of any higher-level language-specific analysis, which leads to complexity and possible unsoundness in the handling of, for example, indirect jumps and stack-allocated arrays. While even they must make the assumption that the code conforms to a "standard compilation model" where a run-time stack of activation records are pushed and popped on function call and return, their approach is more generic out of the box. We instead advocate a clean modularization to enable reuse of decompiler components in order to make customized pipelines more palatable.

Tröger and Cifuentes [27] give a technique to identify virtual method dispatch in machine code binaries based on computing a backward slice from the indirect call. They also try to be generic to any compiler, which necessarily leads to difficulties and imprecision that are not problems for us. Cifuentes *et al.* [10] describe a decompiler from SPARC assembly to C. Driven by the program understanding application, most of their focus is on recovering structured control-flow, which is often unnecessary (if not undesirable) for targeting program analyses.

6 Conclusion and Future Work

We have described a flexible and modular methodology for building assembly code analyses based on a novel notion of cooperating decompilers. We have shown the effectiveness of our framework through three example decompiler pipelines that share low-level components: for the output of `gcc`, `gcj`, and compilers for the Cool object-oriented language.

We are particularly interested in assembly-level analyses for addressing mobile-code safety [25, 23], ideally in a foundational but also practical manner. As such, we have designed our decompilation framework with soundness in mind (e.g., making decompilers work one instruction at a time and working in the framework of abstract interpretation), though we have not yet constructed machine-checkable soundness proofs for our example decompilers. To achieve this, we envision building on our prior work on *certified program analyses* [7], as well as drawing on abstract interpretation-based transformations [16, 26]. Such a modularization of code as we have achieved will likely be critical for feasibly proving the soundness of analysis implementations in a machine-checkable manner. This motivation also partly justifies our use of reflection tables produced by `gcj` or debugging information from `gcc`, as it seems reasonable to trade-off, at least, some annotations for safety checking.

References

[1] A. Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, July 1996.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages (POPL)*, pages 1–11, 1988.

[3] A. W. Appel. Foundational proof-carrying code. In *Logic in Computer Science (LICS)*, pages 247–258, June 2001.

[4] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction (CC)*, pages 5–23, 2004.

[5] G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Computer-Aided Verification (CAV)*, pages 158–163, 2005.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, 2005.

[7] B.-Y. E. Chang, A. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 174–189, 2006.

[8] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. Type-based verification of assembly language for compiler debugging. In *Types in Language Design and Implementation (TLDI)*, pages 91–102, 2005.

[9] B.-Y. E. Chang, M. Harren, and G. C. Necula. Analysis of low-level code using cooperating decompilers. Technical Report EECS-2006-86, UC Berkeley, 2006.

[10] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Software Maintenance (ICSM)*, pages 228–237, 1998.

[11] M. Codish, A. Mulkers, M. Bruynooghe, M. J. G. de la Banda, and M. V. Hermenegildo. Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst.*, 17(1):28–44, 1995.

[12] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Programming Language Design and Implementation (PLDI)*, pages 95–107, 2000.

[13] A. Cortesi, B. L. Charlier, and P. V. Hentenryck. Combinations of abstract domains for logic programming. In *Principles of Programming Languages (POPL)*, pages 227–239, 1994.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 234–252, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[16] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages (POPL)*, pages 178–190, 2002.

[17] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 2002.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Computer-Aided Verification (CAV)*, pages 526–538, 2002.

[19] IDA Pro disassembler. `http://www.datarescue.com/idabase`.

[20] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[21] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *Principles of Programming Languages (POPL)*, pages 270–282, 2002.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[23] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[24] A. Mycroft. Type-based decompilation. In *European Symposium on Programming (ESOP)*, pages 208–223, 1999.

[25] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, pages 106–119, Jan. 1997.

[26] X. Rival. Abstract interpretation-based certification of assembly code. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 41–55, 2003.

[27] J. Tröger and C. Cifuentes. Analysis of virtual method invocation for binary translation. In *Reverse Engineering (WCRE)*, pages 65–74, 2002.

[28] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13, 1999.