SAIL: Static Analysis Intermediate Language with a Two-Level Representation

Isil Dillig Thomas Dillig Alex Aiken
{isil, tdillig, aiken}@cs.stanford.edu

Department of Computer Science
Stanford University

Abstract. In this paper, we present SAIL (Static Analysis Intermediate Language), a front-end which provides both a high- and a low-level representation of programs and maintains a precise mapping from the low-level instructions to the high-level expressions and statements in the original source code. This two-level representation makes it easy to perform semantic analysis of programs on the low-level representation while having the ability to relate this low-level reasoning back to the source code, allowing precise feedback in terms of the original program. SAIL's two-level representation is specifically targeted for program analysis, provides extensive support for control flow graphs and serialization, is built on GCC 4.3.4, and is currently targeted for C code. SAIL is freely available under the BSD license from `http://www.stanford.edu/~isil/sail`.

1 Introduction

When choosing an intermediate language for program analysis, there are two orthogonal, and often competing, considerations: Low-level representations, such as Java bytecode [1] and the LLVM instruction set [2], provide a small number of basic instructions, greatly simplifying the task of performing semantic static analyses. On the other hand, such low-level representations are notoriously difficult to relate back to the original program and lose many of the syntactic clues present in source code. For this reason, high-level intermediate languages, such as CIL [3], are preferable in some contexts, as they enable accurate error-reporting as well as extracting information from high-level program constructs.

However, in many situations, neither only the high-level nor the low-level representation is ideal, and choosing one language over the other compromises a potential benefit of the representation that was not chosen. For example, consider a program analysis framework that performs a semantic analysis (e.g., a pointer analysis), but also wants to check for properties that are definable only through high-level syntactic constructs, such as *"Are expressions with mutual side effects used as arguments to function calls in C?"* or *"Can the lack of a* `break` *statement in some branch of a* `switch` *construct lead to an error?"*. While a low-level representation may greatly simplify the task of performing a semantic analysis, it not only makes error reporting cumbersome but also makes certain properties, such as the ones mentioned above, very difficult, if not impossible, to check.

Therefore, neither only a high-level nor only a low-level representation is ideal for any static analysis task.

In this paper, we present a two-level representation, called Sail (Static Analysis Intermediate Language), that bridges the gap between high- and low-level intermediate representations. Sail provides both a high-level intermediate language that preserves source-level constructs and a low-level language that provides a language-independent representation suitable for semantic analysis. Each instruction in the low-level language allows at most one load or store and has a well-defined mapping to the statement or expression in the high-level language from which it originates. Sail therefore makes it possible to utilize the benefits of both the high- and the low-level representations simultaneously, without needing to switch to a different front-end for different program analysis tasks.

Our current implementation of Sail integrates with GCC 4.3.4, can parse any C program that GCC parses, and is itself written in C++. While Sail's low-level language is expressive enough to model both safe and unsafe imperative languages, it currently only parses C code, with C++ support being under development. In addition to providing a two-level intermediate representation, Sail also provides specialized kinds of control flow graphs, suitable for different styles of program analysis tasks, such as summary-based analysis. In addition, Sail provides a robust and fast serialization framework that allows the intermediate representations and control flow graphs to be efficiently and compactly written to and read from disk. Sail is freely available under the BSD license from `http://www.stanford.edu/~isil/sail`.

The rest of this paper is organized as follows: In Section 2, we describe our low-level intermediate language and discuss why this representation is convenient for semantic analysis tasks. In Section 3, we highlight some aspects of our high-level intermediate language for C and describe benefits and disadvantages of using GCC as a front-end for generating the high-level intermediate representation. In Section 4, we describe stylized variations of control flow graphs that make it easier or more efficient to perform summary-based and path-sensitive analyses. In Section 5, we describe the serialization support Sail provides. Finally, Section 6 reports on our experience using Sail in the Compass program verification framework, and Section 7 surveys related work.

To summarize, this paper makes the following contributions:

– We propose a two-level intermediate language as a way to bridge the gap between high-level and low-level intermediate representations.
– We describe a low-level intermediate language conducive for performing semantic program analysis.
– We describe variations on control flow graphs useful for performing summary-based and path-sensitive program analyses.

2 The Low-Level Representation

As mentioned in Section 1, the main goal of our low-level representation is to simplify semantic analysis by having only a few instructions involving no

more than one load, store, or arithmetic computation per instruction, similar to three-address code. Figure 1 shows the basic instructions of SAIL's low-level language that do not modify control flow. In this figure, we use the notation $v, v_1, \ldots$ to denote variables, and $s, s_1, \ldots$ to denote *symbols*, which can be variables or constants. We use $\boldsymbol{f}$ to denote a non-empty vector of field selectors. A *binop* represents any element in the set $\{+, -, *, /, \%, <, \leq, >, \geq, =, \neq, <<, >>, |, \&, \hat{}, \&\&, ||\}$, and a *unop* denotes any element of $\{!, \tilde{}, -\}$. Note that the $+$ operator can operate on integers, pointers, and floating point numbers.

`Assignment` :	v	$=$	s
`Address` :	v	$=$	$\&s$
`Load` :	v	$=$	$*s$
`Store` :	$*s_1$	$=$	s_2
`Field Read` :	v_1	$=$	$v_2.\boldsymbol{f}$
`Field Write` :	$v_1.\boldsymbol{f}$	$=$	v_2
`Array Read` :	v_1	$=$	$v[s]$
`Array Write` :	$v[s_1]$	$=$	s_2
`Cast` :	v_1	$=$	$(\tau)\ s$
`Binop` :	v	$=$	$s_1\ \mathit{binop}\ s_2$
`Unop` :	v	$=$	$\mathit{unop}\ s$
`Function Call` :	v	$=$	$f(s_1, \ldots, s_n)$
`Function Pointer Call` :	v_1	$=$	$(*v_2)(s_1, \ldots, s_n)$

Fig. 1: Basic low-level instructions not modifying control flow.

The instructions presented in Figure 1 are similar to Java bytecode [1], however to model unsafe languages like C, instructions such as `Address` become necessary. In `Field Read` and `Field Write` instructions, we use a sequence of field selectors rather than a single field selector since the aggregate offset is always known at compile time. In fact, a sequence of field selectors is internally represented as the selectors' net byte offset. In the `Address`, `Load`, and `Store` instructions, observe that symbols rather than variables are used as operands because taking the address of integer and string constants as well as storing into and loading from constants is legal in C.

In the `Array Read` and `Array Write` instructions, the variable v always refers to a non-pointer array. If the variable `a` in the program statement `x` = `a`[2] refers to a pointer array of integers, the following sequence of low-level instructions is generated:

```
t1 = a + 8;
x = *t1;
```

We believe such a representation is desirable because it disambiguates syntactically identical expressions that have very different semantics. Observe that this low-level representation forces us to generate many binop instructions involving

pointer arithmetic that did not appear syntactically in the original source code. Pointer arithmetic arises not only from the use of pointer arrays, but also from many uses of the "address of" operator. For example, the C statement

```
y = &b->f;
```

is translated into the low-level language as:

```
y = b + offset(f)
```

One consequence of such a low-level language from the perspective of a program analysis system is that pointer arithmetic cannot be ignored when reasoning about programs. We believe this to be a benefit since pointer arithmetic is common enough in real C code that it cannot realistically be ignored when performing program analysis, and this low-level representation allows many syntactically distinct high-level constructs to be treated uniformly. Another advantage of this representation is that it relieves the program analysis designer from the burden of correctly interpreting the semantics of complex expressions such as `&a[3].x->g` since the semantics of the corresponding low-level instructions are straightforward and unambiguous. In our experience, this low-level uniform representation is instrumental in avoiding soundness holes in static analyses that result from incorrectly interpreting the semantics of C constructs while implementing an analysis.

The low-level language of Sail is typed, and types τ are defined according to the following grammar:

$$\begin{aligned}
\tau &:= \text{void} \mid \alpha \mid \mathit{ptr}(\tau) \mid \mathit{array}(\tau) \\
&\quad \mid \mathit{struct}(o_1 : \tau_1, \ldots o_k : \tau_k) \\
&\quad \mid \mathit{union}(\tau_1, \ldots \tau_k) \\
&\quad \mid (\tau_1 \times \ldots \times \tau_n) \rightarrow \tau_{ret} \\
\alpha &:= \kappa \times \mathit{size} \times \sigma \\
\kappa &:= \text{IEEE} \mid \text{binary} \\
\sigma &:= \text{signed} \mid \text{unsigned}
\end{aligned}$$

In this grammar, a base type α is a triple consisting of a kind κ, a size specifying the number of bytes, and a sign value σ indicating whether this value is to be interpreted as signed or unsigned. The kind κ determines whether the bit pattern of this value is to be interpreted as an IEEE floating point number or as binary (integer) encoding. A type τ can be void, a base type α, a pointer to a type τ, a (non-pointer) array whose elements are of type τ, a struct mapping offsets to types τ_i, a union with possible types $\tau_1, \ldots \tau_k$, or a function type mapping arguments with types $\tau_1, \ldots \tau_n$ to return type τ_{ret}. Accessing a union is modeled by casting the union type to its selected element type, and vararg functions are identified by a qualifier on the function type.

Typing rules for the instructions from Figure 1 are given in Figure 2. Here, we use the notation $\mathit{offset}(\tau, f)$ to denote the offset of field f in type τ. We also use Int as an abbreviation for any base type that has kind binary, and Float for any base type with kind IEEE.

Assign

$$\frac{\mathit{\Gamma} \vdash v : \tau \quad \mathit{\Gamma} \vdash s : \tau}{\mathit{\Gamma} \vdash v = s}$$

Address

$$\frac{\mathit{\Gamma} \vdash v : \mathit{ptr}(\tau) \quad \mathit{\Gamma} \vdash s : \tau}{\mathit{\Gamma} \vdash v = \&s}$$

Load

$$\frac{\mathit{\Gamma} \vdash v : \tau \quad \mathit{\Gamma} \vdash s : \mathit{ptr}(\tau)}{\mathit{\Gamma} \vdash v = *s}$$

Store

$$\frac{\mathit{\Gamma} \vdash s_1 : \mathit{ptr}(\tau) \quad \mathit{\Gamma} \vdash s_2 : \tau}{\mathit{\Gamma} \vdash *s_1 = s_2}$$

Cast

$$\frac{\mathit{\Gamma} \vdash v : \tau}{\mathit{\Gamma} \vdash v = (\tau)s}$$

FieldRead

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v_1 : \tau_i \\ \mathit{\Gamma} \vdash v_2 : \tau \\ \tau = \mathit{struct}(o_1 : \tau_1, \ldots o_i : \tau_i \ldots o_k : \tau_k) \\ o_i = \mathit{offset}(\tau, f)\end{array}}{\mathit{\Gamma} \vdash v_1 = v_2.f}$$

FieldWrite

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash s : \tau_i \\ \mathit{\Gamma} \vdash v_1 : \tau \\ \tau = \mathit{struct}(o_1 : \tau_1, \ldots o_i : \tau_i \ldots o_k : \tau_k) \\ o_i = \mathit{offset}(\tau, f)\end{array}}{\mathit{\Gamma} \vdash v.f = s}$$

ArrayRead

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v_1 : \tau \\ \mathit{\Gamma} \vdash s : \text{Int} \\ \mathit{\Gamma} \vdash v_2 : \mathit{array}(\tau)\end{array}}{\mathit{\Gamma} \vdash v_1 = v_2[s]}$$

ArrayWrite

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v : \mathit{array}(\tau) \\ \mathit{\Gamma} \vdash s_1 : \text{Int} \\ \mathit{\Gamma} \vdash s_2 : \tau\end{array}}{\mathit{\Gamma} \vdash v[s_1] = s_2}$$

Unop

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v : \tau \\ \mathit{\Gamma} \vdash s : \tau \\ \tau \in \begin{cases}\{\text{Int}, \text{Float}\} & \mathit{if\ unop\ is\ -} \\ \{\text{Int}\} & \mathit{otherwise}\end{cases}\end{array}}{\mathit{\Gamma} \vdash v = \mathit{unop}\ s}$$

Binop

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v : \tau \\ \mathit{\Gamma} \vdash s_1 : \tau \\ \mathit{\Gamma} \vdash s_2 : \begin{cases}\text{Int} & \mathit{if}\ \tau = \mathit{ptr}(\tau') \\ \tau & \mathit{otherwise}\end{cases} \\ \tau \in \begin{cases}\{\text{Int}, \text{Float}, \mathit{ptr}(\tau')\} & \mathit{if}\ \text{binop is} + \\ \{\text{Int}\} & \mathit{if}\ \text{binop is} \ \%, <<, >>, |, \&, \hat{}, \&\&, || \\ \{\text{Int}, \text{Float}\} & \mathit{otherwise}\end{cases}\end{array}}{\mathit{\Gamma} \vdash v = s_1\ \text{binop}\ \text{s}_2}$$

FunctionCall

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash f : (\tau_1 \times \ldots \times \tau_n) \rightarrow \tau_{ret} \\ \mathit{\Gamma} \vdash s_1 : \tau_1, \ldots, s_n : \tau_n \\ \mathit{\Gamma} \vdash v : \tau_{ret}\end{array}}{\mathit{\Gamma} \vdash v = f(s_1, \ldots, s_n)}$$

Function Pointer Call

$$\frac{\begin{array}{l}\mathit{\Gamma} \vdash v_2 : \mathit{ptr}((\tau_1 \times \ldots \times \tau_n) \rightarrow \tau_{ret}) \\ \mathit{\Gamma} \vdash s_1 : \tau_1, \ldots, s_n : \tau_n \\ \mathit{\Gamma} \vdash v : \tau_{ret}\end{array}}{\mathit{\Gamma} \vdash v_1 = (*v_2)(s_1, \ldots, s_n)}$$

Fig. 2: Typing Rules for the Low-Level Intermediate Language

The low-level types mirror the low-level intermediate language by exclusively focusing on the physical layout of data in memory. We believe this to be an advantage in performing sound program analysis of unsafe languages such as C, which treat types as polite suggestions, making it unrealistic to rely on the type safety of the original program for verification. However, our low-level type representation makes it easy to check forms of type consistency, for example, using techniques like *physical subtyping* [4].

Figure 3 presents the three control instructions in the low-level language. Here, the only unusual feature is the branch instruction, which allows for arbitrary fan-out. SAIL guarantees that exactly one of the s_i's is non-zero in any evaluation, i.e., $s_i \neq 0$ and $s_j \neq 0$ are mutually exclusive for any $i \neq j$, and $\bigvee_i s_i \neq 0 = true$. We discuss the usefulness of this representation in Section 4.2.

`Branch` : $\{(s_1 \neq 0 \ ?\ l_1), (s_2 \neq 0 \ ?\ l_2), \ldots (s_n \neq 0 \ ?\ l_n)\}$
`Jump` : $goto\ l$
`Label` : l

Fig. 3: Low-level instructions modifying control flow.

In total, the low-level language contains 22 instructions, including one for inline assembly, one needed for the GCC address of label extension as well as an `assume` and `static_assert` instruction useful for program analysis. Observe that SAIL has no declarations; all variables are implicitly declared the first time they are used. There is also no notion of nested scope within a function; all variable names are syntactically disambiguated by renaming them where necessary.

3 The High-Level Representation

The high-level intermediate representation of SAIL is effectively a disambiguated and type-checked abstract syntax tree. The high-level representation aims to preserve all relevant syntactic information present in the original source code and is therefore language-specific. Figures 4 and 5 present the grammar for statements and expressions in the high-level language for C. Since most constructs in this grammar are standard for C, we do not discuss the grammar in detail. Observe that this grammar preserves high-level constructs like the post-increment unop expression (e.g., `i++`) or the expression list expression (e.g., `i++, j--`). While this language is not conducive for performing program analysis, it retains a close and direct connection to the source code and allows for meaningful error reporting as well as checking for particular constructs used in the original source code.

```
stmt := SetStmt(exp, exp)
     |  IfStmt(exp, stmt, stmt)
     |  ReturnStmt(exp option)
     |  SwitchStmt(exp, stmt)
     |  ExpStmt(exp)
     |  ForLoop(exp option, exp option, exp option, stmt)
     |  WhileLoop(exp, stmt)
     |  DoWhileLoop(exp, stmt)
     |  BreakStmt
     |  Continue
     |  Label(id)
     |  GotoStmt(id)
     |  AssemblyStmt(string, exp list, exp list)
     |  BlockStmt(var_decl list, stmt list)
```

Fig. 4: Statements in the High Level Intermediate Language

```
exp := AddressExp(exp, type)
    |  ArrayRefExp(exp, exp, type)
    |  BinopExp(exp, exp, binop_type, type)
    |  UnopExp(exp, unop_type, type)
    |  BlockExp(BlockStmt, type)
    |  CastExp(exp, type)
    |  ConditionalExp(exp, exp, exp, type)
    |  DerefExp(exp, type)
    |  ExpListExp(exp list, type)
    |  FieldRefExp(exp, name, offset, type)
    |  FunctionAddressExp(id, type)
    |  FunctionCallExp(id, exp list, type)
    |  FunctionPtrCallExp(exp, exp list, type)
    |  ModifyExp(SetStmt, type)
    |  VariableExp(id, type)
    |  IntConstExp(int, type)
    |  RealConstExp(float, type)
    |  StringConstExp(string, type)
```

Fig. 5: Expressions in the High-Level Intermediate Language

```
key.c:key_to_blob {
   __temp1 = 0 == key;
   __temp2 = !__temp1;
   if(__temp1) then goto __label1 else goto __label2;
   __label1;
   __temp3 = &\"key_to_blob: key == NULL\";
   __temp4 = (u_char*) __temp3;
   __temp5 = error(__temp4);
   __return = 0;
   goto __return_label;
   goto __label2;
   __label2;
   __temp6 = & (b);
   __temp7 = buffer_init(__temp6);
   __temp8 = key->type;
   __temp9 = __temp8 == 2;
   __temp10 = __temp8 == 1;
   __temp11 = 1 <= __temp8;
   __temp12 = __temp8 <= 2;
   __temp13 = __temp11 && __temp12;
   __temp14 = !__temp13;
   switch (<__temp9 => case 2: >
           <__temp10 => case 1: >
           <__temp14 => default: >);
   case 2: ;
   __temp15 = & (b);
   __temp16 = key_ssh_name(key);
   __temp17 = buffer_put_cstring(__temp15, __temp16);
   __temp18 = & (b);
   __temp19 = key->dsa;
   __temp20 = __temp19->p;
   __temp21 = buffer_put_bignum2(__temp18, __temp20);
   __temp22 = & (b);
   __temp23 = key->dsa;
   __temp24 = __temp23->q;
   __temp25 = buffer_put_bignum2(__temp22, __temp24);
   __temp26 = & (b);
   __temp27 = key->dsa;
   __temp28 = __temp27->g;
   __temp29 = buffer_put_bignum2(__temp26, __temp28);
   __temp30 = & (b);
   __temp31 = key->dsa;
   __temp32 = __temp31->pub_key;
   __temp33 = buffer_put_bignum2(__temp30, __temp32);
   goto __label1;
   case 1: ;
   __temp34 = & (b);
   __temp35 = key_ssh_name(key);
   __temp36 = buffer_put_cstring(__temp34, __temp35);
   __temp37 = & (b);
   __temp38 = key->rsa;
   __temp39 = __temp38->e;
   __temp40 = buffer_put_bignum2(__temp37, __temp39);
   __temp41 = & (b);
   __temp42 = key->rsa;
   __temp43 = __temp42->n;
   __temp44 = buffer_put_bignum2(__temp41, __temp43);
   goto __label1;
   default: ;
   __temp45 = &\"key_to_blob: unsupported key type %d\";
   __temp46 = (u_char*) __temp45;
   __temp47 = key->type;
   __temp48 = error(__temp46, __temp47);
   __temp49 = & (b);
   __temp50 = buffer_free(__temp49);
   __return = 0;
   goto __return_label;
   __label1;
   __temp51 = & (b);
   __temp52 = buffer_len(__temp51);
   len = __temp52;
   __temp53 = lenp != 0;
   __temp54 = !__temp53;
   if(__temp53) then goto __label3 else goto __label4;
   __label3;
   *lenp = len;
   goto __label4;
   __label4;
   __temp55 = blobp != 0;
   __temp56 = !__temp55;
   if(__temp55) then goto __label5 else goto __label6;
   __label5;
   __temp57 = xmalloc(len);
   __temp58 = (long int) __temp57;
   __temp59 = (u_char*) __temp58;
   *blobp = __temp59;
   __temp60 = *blobp;
   __temp61 = (void*) __temp60;
   __temp62 = & (b);
   __temp63 = buffer_ptr(__temp62);
   __temp64 = (long int) __temp63;
   __temp65 = (void*) __temp64;
   __temp66 = (long unsigned int) len;
   __temp67 = memcpy(__temp61, __temp65, __temp66);
   goto __label6;
   __label6;
   __temp68 = & (b);
   __temp69 = buffer_ptr(__temp68);
   __temp70 = (long int) __temp69;
   __temp71 = (void*) __temp70;
   __temp72 = (long unsigned int) len;
   __temp73 = memset(__temp71, 0, __temp72);
   __temp74 = & (b);
   __temp75 = buffer_free(__temp74);
   __return = len;
   goto __return_label;
   __return_label;
}
```

Fig. 6: The low-level representation of the `key_to_blob` function from OpenSSH. Observe that, in contrast to the presentation in Section 2, the load and store instructions are allowed to use an optional offset. The reason for this extension is discussed later in Section 6.

```
static int key_to_blob(struct Key* key, u_char** blobp, int* lenp, ...)
{
   struct Buffer b;
   int len;
   if((key)==(0)) {
       error((u_char*)&"key_to_blob: key == NULL");
       return 0;
   }
   buffer_init(&b);
   switch(*(key).type) {
       case 2:
       buffer_put_cstring(&b, key_ssh_name(key));
       buffer_put_bignum2(&b, *(*(key).dsa).p);
       buffer_put_bignum2(&b, *(*(key).dsa).q);
       buffer_put_bignum2(&b, *(*(key).dsa).g);
       buffer_put_bignum2(&b, *(*(key).dsa).pub_key);
       break;

       case 1:
       buffer_put_cstring(&b, key_ssh_name(key));
       buffer_put_bignum2(&b, *(*(key).rsa).e);
       buffer_put_bignum2(&b, *(*(key).rsa).n);
       break;

       default:
       error((u_char*)&"key_to_blob: unsupported key type %d", *(key).type);
       buffer_free(&b);
       return 0;
   }
   len = buffer_len(&b);
   if((lenp)!=(0)) *(lenp) = len; ;
   if((blobp)!=(0)) {
        *(blobp) = (u_char*)(long int)xmalloc(len);
        memcpy((void*)*(blobp), (void*)(long int)buffer_ptr(&b), (long unsigned int)len);
   }
   memset((void*)(long int)buffer_ptr(&b), 0, (long unsigned int)len);
   buffer_free(&b);
   return len;
}
```

Fig. 7: The mapping from the low-level representation back to the high-level representation is illustrated by printing the high-level representation from the low-level language. This is done by going through each instruction in the low-level language and printing the corresponding high-level statement; low-level instructions that are associated with an expression are skipped.

Every statement in the high-level language corresponds to at least one instruction in the low-level language. Furthermore, an expression used in the high-level language may also generate additional instructions using temporary variables in the low-level language. For instance, an expression such as `**x` which involves two loads, requires the introduction of a temporary variable used to capture the result of the first memory access. When translating the high-level language to the low-level representation, each instruction in the low-level representation is associated with the original expression or statement that generated it. For example, if `a=f(**x)` is a statement appearing in the high-level language, then the generated low-level instructions `t1 = *x`, `t2 = *t1`, and `a=f(t2)` are associated with the expressions `*x`, `**x`, and the statement `a=f(**x)` respectively.

To illustrate the mapping between the low- and high-level representations, Figure 6 shows the low-level representation of a function called `key_to_blob` from OpenSSH, and Figure 7 illustrates how the mapping between the two representations allows for printing the high-level representation from the low-level instructions. As is evident from the code in Figure 6, while the low-level representation may be very convenient for doing analysis, it is extremely unwieldy for relating the analysis back to the source code. The precise mapping between the two representations makes it possible to reason about the program using the low-level language while still being able to relate this reasoning back to the original source code.

3.1 Using GCC to Generate the High-Level Representation

In this section, we report on our experience using GCC as a front-end for Sail. Our search for a front-end was guided by the following considerations:

- The front-end should be able to preprocess, parse, and type-check all available open-source programs.
- It should expose a representation that is high-level enough to construct Sail's high-level intermediate language.
- Parsing the source code should be fast, i.e., it should not take longer than compilation.
- The front-end should support multiple imperative languages, such as C, C++, and Java, to make Sail easily extensible.
- The front-end should be open-source and under active development.

These design constraints led us directly to use GCC 4.3.4 as a front-end for Sail. Since almost all C-based open-source projects use GCC as their compiler, GCC is able to preprocess and parse all applications we are interested in analyzing. Furthermore, while not ideal, GCC does expose a reasonably high-level representation, and since GCC is an industrial-strength compiler, parsing and type-checking are very fast. In addition, to the best of our knowledge, GCC is the only open-source front-end that parses a wide variety of languages, such as C, C++, FORTRAN, Java, Objective C etc.

In our experience, interfacing with GCC is a relatively straightforward task, and SAIL adds less than three thousand lines of code to GCC to generate its high-level language. While the internal representation of GCC is not particularly well-documented, perusing the source files reveals a reasonably clean and usable internal representation. In our experience, the only real disadvantage of using GCC as a front-end is that some program transformations are performed during parsing, making it very difficult to recover as high-level a representation as we would like. Specifically, GCC replaces all looping constructs with goto statements and labels during parsing. As a result, while our high-level representation is designed to preserve syntactic looping constructs, it currently does not. However, we feel that the benefits of using GCC outweigh this disadvantage.

In addition to fulfilling our initial design goals, using GCC has other valuable benefits for performing program analysis. These include automatic and accurate identification of memory allocators and exit functions, i.e., functions that abort execution. Furthermore, GCC provides byte offsets for struct fields, making it easy to construct SAIL's low-level type representation.

4 Stylized Control Flow Graphs

Since control flow graphs are fundamental to many program analyses, SAIL provides extensive support for CFG construction. Although rare, irreducible control flow graphs do arise in some applications, such as the Linux kernel; therefore, SAIL supports transforming irreducible control flow graphs to reducible ones using node-splitting based on T1-T2 transformations [5]. Since SAIL knows which functions abort execution, calls to exit functions modify control flow. If a basic block B contains a call to an exit function, the successor of B is always a special block, called the *exception block*. In addition, SAIL provides two extensions to the standard control flow graph that aid summary-based analysis and path-sensitive analysis respectively.

4.1 Summary CFG

The goal of summary-based analysis is to generate *summaries* of functions and loop bodies that encode the relevant behavior of these *summary units* with respect to some property and independent of the context in which they appear. When a function call or loop is encountered during the analysis, the summary associated with this summary unit is retrieved and *instantiated* (i.e., applied), potentially to a fixed-point. Polymorphic summary-based analysis has the benefit of being naturally context-sensitive and allows irrelevant information to be discarded at summarization points. Furthermore, summary-based analysis allows for local one-function (or one-loop) at a time reasoning and is often key to scalability [6–9].

To perform summary-based analysis, it is necessary to identify an entry and exit point in the control flow graph, delimiting each summary unit. While this task is easy for functions by connecting all exit blocks to a single exit block,

this task is more involved for loops. To be concrete, consider the following code example from Figure 8. The standard control flow graph associated with `foo`

```
void foo(int* a, int size, int elem)
{
   int i;
   int found = 0;
   for(i=0; i<size; i++) {
       if(a[i] == elem) {
           found = 1;
           break;
       }
   }
}
```

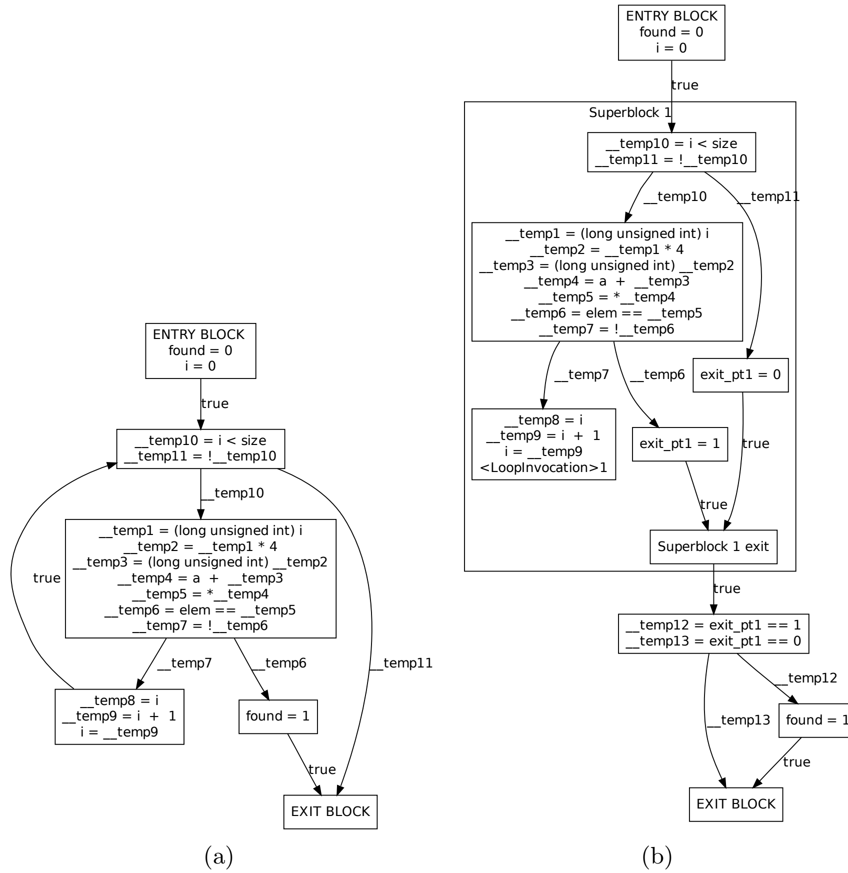
Fig. 8: Example illustrating multiple exit points for loops

is shown in Figure 9a. Here, observe that the natural loop in `foo` has two exit points reaching two different blocks in the body of `foo` because the statement `found = 1` is *not* part of the natural loop (as the loop header is unreachable from the statement), even though it is part of the syntactic looping construct. Such control flow makes it difficult to generate and instantiate summaries since there is no unique point where the summary associated with the loop can be generated or instantiated.

To make summary-based analysis easier, SAIL generates *summary CFG's* where all loops are explicitly marked as summary units using *superblocks* which always have unique entry and exit points and have no explicit back-edges. Figure 9b shows the summary CFG for `foo`. Here, observe that the summary unit associated with the loop is explicitly marked as a superblock, indicated by the rectangular box. This superblock has exactly one exit block marking where the loop summary should be generated. To make this transformation possible, SAIL introduces a temporary variable, called `exit_pt1` in Figure 9b, that encodes which exit point was taken. The basic block following the superblock branches on the value of `exit_pt1` to faithfully encode the semantics of the original function. Also, note that, while the superblock no longer has a back edge, a *loop invocation instruction* models the loop as a tail recursive function. In our experience, this summary CFG representation makes implementing summary-based analyses significantly easier.

4.2 Multi-branch CFG

SAIL generates *multi-branch* control flow graphs that allow a basic block to have an arbitrary number of successors. This representation allows for a much more compact encoding of `switch` statements and can be very beneficial in path-sensitive analyses by dramatically reducing the size of constraints used to encode



(a) (b)

Fig. 9: The standard CFG (on the left) and the summary CFG (on the right)

path conditions. This is the case because the conditions in a `switch` statement are, by construction, disjoint, and restricting basic blocks to have at most two successors is equivalent to enumerating a redundant `if-then-else` structure.

To be concrete, consider the function from Figure 10 which uses a `switch` statement. Figure 11a shows the standard control flow graph with at most two successors per block, while Figure 11b shows the multi-branch CFG. Observe that the multi-branch CFG is much more compact than the standard CFG, and this difference becomes more pronounced as the number of case labels increases. To understand the potential impact of this representation for a path-sensitive analysis, consider computing the *statement guard* for program point `(*)` in Figure 10, which encodes the constraint under which this program point is reached. Using the standard CFG shown in Figure 11a, the statement guard at point `(*)`

```
void bar(unsigned int x)
{
    int a;
    switch(x)
    {
        case 0: a = 0; break; case 1: a = 1; break;
        case 2: a = 2; break; case 3: a = 3; break;
        case 4: a = 4; break; case 5: a = 5; (*)  break;
        default: a = -1;
    }
}
```

Fig. 10: Example illustrating benefit of multi-branch CFGs.

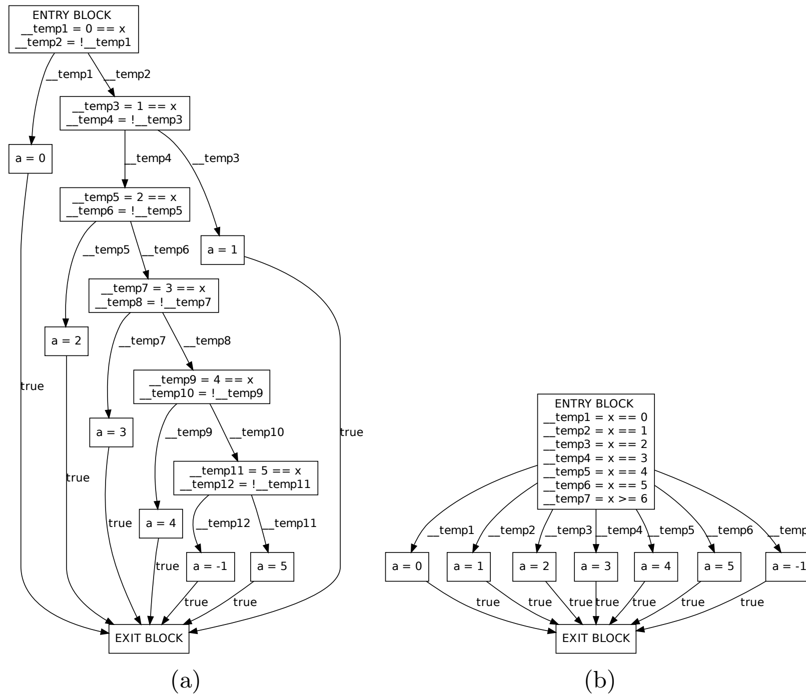


Fig. 11: Multi-branch CFG (on the right) and the standard CFG with only two branches (on the left)

is computed as:

$$x \neq 0 \wedge x \neq 1 \wedge x \neq 2 \wedge x \neq 3 \wedge x \neq 4 \wedge x = 5$$

whereas the statement guard using the CFG from Figure 11b is just $x = 5$. In the authors' previous experience using the Saturn program analysis system [10],

restricting the control flow graph to have at most two successors can be a source of scalability problems when performing some kinds of path-sensitive analysis.

5 Serialization Support

Sail supports writing and reading the intermediate language representations and control flow graphs to and from disk. This serialization mechanism allows for analyzing more than one translation unit at a time, which is not possible without serialization since Sail is invoked on one translation unit. Sail creates one file on disk per function encountered during parsing. Since some functions may appear in many translation units, e.g., functions implemented in header files, Sail automatically detects functions that have already been parsed and does not create duplicate files.

The ability to serialize one function at a time has two important advantages: First, if a source file is edited, the entire intermediate representation of the program can be updated by recompiling only the translation units this file belongs to. This is beneficial when performing program analysis on a large application because local changes, such as adding an annotation or commenting out statements for debugging, do not require reparsing the entire application. The second advantage of serializing one function at a time is that analyses can load only those functions into memory that are currently being analyzed and does not require keeping the intermediate representation of the entire application resident in memory.

For speed and space efficiency, Sail uses a binary format to serialize data. Earlier versions of Sail that utilized an XML-based encoding resulted in much larger data sets (even after compression) as well as much slower reading from and writing to disk, making binary encoding a more practical alternative. Using the binary encoding, all the intermediate representation files (including control flow graphs) of OpenSSH take less than 20 MB and can be reconstructed into memory in their entirety in less than one tenth of a second.

6 Experience using SAIL

Sail was developed as a front-end for the Compass program verification framework for analyzing C programs. Compass performs a very precise, path- and context-sensitive pointer and value analysis and also tracks contents of arrays. The low-level intermediate language of Sail was extremely beneficial in developing these analyses for two reasons: First, since the low-level language only allows for basic instructions that involve no more than one load, store, or arithmetic operation, there is no need to reason about complex expressions when implementing an analysis. Second, the low-level type representation of Sail makes it possible to easily check for type consistency without relying on the type safety of the original program.

One disadvantage of the low-level language presented in Section 2 is that it generates temporaries that correspond to deep copies of structs that are not

present in the original source code. For instance, consider the following C statement:

```
int x = a->f;
```

The translation of this statement to the low-level language presented in Section 2 is as follows:

```
t1 = *a
x = t1.f
```

Here, notice that `t1` is a deep copy of the struct pointed to by `a` even though no copies are made in the original code. Since our analysis precisely models struct fields, making large numbers of deep copies of structs can adversely affect analysis performance. For this reason, SAIL supports optional offsets for load, store, array read, and array write instructions, and we use SAIL with this option enabled when performing program analysis.

7 Related Work

Many high- and low-level intermediate representations have been proposed for compilers and interpreters. Examples include the SIMPLE intermediate language in the McCAT compiler framework [11], GENERIC and GIMPLE representations used in GCC [12], Jave Bytecode [1] and Microsoft's CIL (Common Intermediate Language), formerly known as MSIL [13]. However, all of these representations are either too low-level to relate back to source code or too high-level to be suitable for analyzing semantic properties of programs.

A series of front-ends have been recently developed to aid program analysis. The CIL (C Intermediate Language) tool developed by Necula et al. provides a high-level yet disambiguated representation of C programs [3]. In contrast to CIL, SAIL provides both a high and a low-level representation; however, SAIL's high-level representation is much higher than CIL whereas its low-level representation is much lower. While CIL allows for intricate program transformations, SAIL is exclusively targeted for performing static analysis. In contrast to CIL which uses its own parser, SAIL builds on GCC and therefore parses any program that GCC parses. Furthermore, by building on GCC, SAIL does not interfere with the build process of applications, does not require manual preprocessing of files, and does not require the entire program to be reparsed when a source file is edited. In addition, while CIL can be significantly slower than a full compilation, SAIL adds less than 10% overhead to compilation.

The SUIF compiler infrastructure [14] provides a series of intermediate representations suitable for different levels of program optimization and transformations for C and C++. Unlike SAIL, the main focus of the SUIF compiler infrastructure is not an accurate mapping between different levels of intermediate representations. The SUIF infrastructure does not support many GNU C extensions that appear in several open source applications, and to the best knowledge of the authors, it is not under active development since 1999.

The Microsoft Phoenix compiler framework [15] is based on the Microsoft MSVC compiler and provides an intermediate language for program analysis. In contrast to Sail, it does not maintain high and low-level representations with a well-defined mapping between the two languages. Furthermore, the Phoenix framework only parses programs that MSVC can parse, and is therefore not suitable for analyzing many open source programs.

The LLVM compilation framework [2] aims to develop a new backend for compiler optimizations. LLVM provides a very low-level intermediate language, similar in spirit to the low-level language of Sail. However, LLVM does not provide any higher level representation, making it impossible to relate reasoning at the low-level back to the original source code. Like Sail, LLVM also provides a front-end based on GCC.

References

1. Yellin, F., Lindholm, T.: The JavaTM Virtual Machine Specification. Addison-Wesley (1999)
2. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society Washington, DC, USA (2004)
3. Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction. (2002) 213–228
4. Chandra, S., Reps, T.: Physical Type Checking for C. SIGSOFT Softw. Eng. Notes **24**(5) (1999) 66–75
5. Hecht, M., Ullman, J.: Flow graph reducibility. In: Proceedings of the fourth annual ACM symposium on Theory of computing, ACM New York, NY, USA (1972) 238–250
6. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Volume 40., ACM New York, NY, USA (2005) 351–363
7. Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2008) 270–280
8. Nystrom, E., Kim, H., Hwu, W.: Bottom-up and top-down context-sensitive summary-based pointer analysis. In: Static Analysis Symposium, Springer (2004) 165–180
9. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2008) 221–234
10. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the Saturn project. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, ACM New York, NY, USA (2007) 43–48

11. Hendren, L., Gao, G., Sridharan, B.: Designing the McCAT compiler based on a family of structured intermediate representations. In: Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, number 757 in Lecture Notes in Computer Science. (1992)
12. GCC: http://gcc.gnu.org/
13. Microsoft: Common Language Infrastructure (CLI): Partition III: CIL Instruction Set. Technical report, ECMA (2002)
14. Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S., Tseng, C., Hall, M., Lam, M., et al.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. ACM Sigplan Notices **29**(12) (1994) 31–37
15. Phoenix: https://connect.microsoft.com/phoenix