KT Advance Reference Manual

Kestrel Technology, LLC

November 1, 2017

Contents

1 Overview

The KT Advance C Analyzer consists of three components:

1. **Parser:** A Mac/Linux executable that takes as input a preprocessed C source file and produces a set of xml files that precisely represent the semantics of the C source file;
2. **C Analyzer:** A Mac/Linux executable that takes as input the semantics files produced by the parser as well as analysis results files if available, and produces a set of xml files that hold analysis results. The C Analyzer will be wrapped in a license manager to protect the contained intellectual property.
3. **PyAdvance:** Python code, provided as source code to licensed users, that performs linking and provides various analyzer invocation, integration, and reporting services.

2 Approach

2.1 Proof by Induction

The goal of the KT Advance C Analyzer is to mathematically prove absence of memory safety vulnerabilities, or, more precisely, prove absence of undefined behavior related to memory safety. The technique used to accomplish this is proof by structural induction on the control flow graph of each function: for each instruction we assume that the state of an execution (viewed as a sequence of states) reaching this point is well-defined (the inductive hypothesis), and we have to prove that all possible states reached after the instruction are still well defined (the inductive step). If indeed we can show that every instruction, starting from a well-defined state, ends in a well-defined state, we can conclude that no undefined behavior is possible in any execution of the application.

2.2 Primary Proof Obligations

The inductive step requires proving for each instruction that the weakest precondition of that instruction with respect to well-definedness as defined in the C Standard [**?**] is program-valid, that is, it is valid on all possible states that an execution can be in when reaching that instruction. We have chosen to express these weakest preconditions as conjunctions of a collection of primitive predicates rather than as one monolithic predicate to facilitate the use of different analysis domains and potentially specialized proof tools for different predicates. We call the primitive predicates that form the weakest precondition "primary proof obligations." For example, the primary proof obligations (also referred to as ppo's) for the instruction `j = j/i` would be that `j` and `i` both be initialized and that `i` is not equal to zero, or, formally

$initialized(i) \wedge initialized(j) \wedge not - zero(i)$, all of which can be proven separately. A list and detailed description of all primitive predicates used is given below.

2.3 Proof Obligation Predicates

allocation-base (ptr:exp): The value of the ptr expression is the address of a dynamically allocated memory region.

common-base (ptr1:exp, ptr2:exp): The value of the ptr1 expression and the value of the ptr2 expression point to the same memory region (where a region can be an allocated region, or a region defined by a declared variable).

common-base-type (ptr1:exp, ptr2:exp): The value of the ptr1 expression and the value of the ptr2 expression point at elements of the same array.

index-lower-bound (index:exp): The value of the index expression is greater than or equal to zero.

index-upper-bound (index:exp, size:exp): The value of the index expression is less than the value of the size expression.

initialized (lhs:lval): The value in the location denoted by lhs is initialized.

initialized-range (ptr:exp, len:exp): The value of the ptr expression points to a memory region of which at least len bytes are initialized starting from the address in ptr.

lower-bound (ptr:exp): The value of the ptr expression is greater than or equal to the lower bound of the memory region it is pointing at (vacuously true for NULL).

non-negative (scalar:exp): The value of the scalar expression is non-negative.

no-overlap (ptr1:exp, ptr2:exp): the value of the ptr1 expression and the value of the ptr2 expression do not point at the same memory region.

not-null (ptr:exp): The value of the ptr expression is not NULL.

not-zero (scalar:exp): The value of the scalar expression is not zero.

null (ptr:exp): The value of the ptr expression is NULL.

null-terminated (ptr:exp): The ptr expression points at a memory region that contains a null-terminator within its bounds.

ptr-lower-bound (t:typ, op:binop, ptr:exp, scalar:exp) The result of the operation op performed on the ptr expression and scalar expression is greater than or equal to the lower bound of the memory region pointed to by the ptr expression.

ptr-upper-bound (t:typ, op:binop, ptr:exp, scalar:exp) The result of the operation op performed on the ptr expression and scalar expression is less than or equal to the upper bound of the memory region pointed to by the ptr expression.

upper-bound (ptr:exp): The value of the ptr expression is less than or equal to the upper bound of the memory region it is is pointing at (vacuously true for NULL).

valid-mem (ptr:exp): The value of the ptr expression is the address of or inside a valid memory region, that is, a memory region that has not been freed.

The following predicates are not a weakest precondition for undefined behavior, but are included to identify constructs generally considered undesirable.

format-string (ptr:exp): The value of the ptr expression points at a string literal.

3 Files

Analysis is modular: the C Analyzer analyzes (preprocessed) c source files in isolation; the PyAdvance integrator transfers results from one file to another in terms of api assumptions, postcondition guarantees, etc. These analysis artifacts are saved in xml files associated with

the entire application, individual files or single functions. Below we list these files, along with their role in the analysis and their format and contents.

All analysis artifacts are kept in a subdirectory `semantics/ktadvance` of the analysis directory. Below we will refer to this directory as the top directory.

3.1 Application-level

The following two files combine information about the entire application at the top level of the

- **target_files.xml:** a list of the source files included in the application.
  created by: Parser
  updated by: none
- **globaldefinitions.xml:** a dictionary of global definitions and declarations that are shared by all source files.
  created by: PyAdvance Linker (advance/linker)
  updated by: none

3.2 File-level

The following files are kept for each source file <f>.c in a directory relative to the top directory that corresponds to their location in the original application source directory.

- **<f>_cfile.xml:** Global definitions.
  created by: Parser
  updated by: none
- **<f>_cdict.xml:** Dictionary of types, variables, expressions, etc. that appear in the program.
  created by: Parser
  updated by: C Analyzer (primary proof obligation generation)
- **<f>_gxrefs.xml:** Mapping between global indices and file-local indices for struct definitions and global variables.
  created by: PyAdvance Linker
  updated by: none
- **<f>_ctxt.xml:** Dictionary of precise locations in the program, expressed as program contexts, a pair of a cfg-context, specifying the location in terms of control-flow-graph nodes, and an exp-context, specifying a location within an expression, in terms of nodes in the syntax tree.

created by: C Analyzer (primary proof obligation generation).
updated by: none

- **<f>_ixf.xml:** Dictionary of components of interface expressions, such as function preconditions, postconditions, and side effects.
  created by: C Analyzer (invariant generation)
  updated by: C Analyzer (invariant generation)
- **<f>_prd.xml:** Dictionary of predicates used in primary and supporting proof obligations.
  created by: C Analyzer (primary proof obligation generator)
  updated by: PyAdvance (creation of supporting proof obligations)

3.3 Function-level

The following files are kept for each function <ff>in file <f>.c, in a subdirectory <f>in the directory that holds the <f>_xxx.xml files.

- **<ff>_cfun.xml:** Complete semantics of the function, expressed as CIL-like data structures, using dictionary indices for types, expressions, etc.
  created by: Parser
  updated by: none
- **<ff>_api.xml:** Application interface artifacts for a function, including assumptions on arguments (used to create supporting proof obligations), postcondition guarantees provided by the function, postcondition requests from other functions.
  created by: C Analyzer (primary proof obligation generation)
  updated by: C Analyzer (invariant gernation, proof obligation check), PyAdvance
- **<ff>_ppo.xml:** Primary proof obligations for a function.
  created by: C Analyzer (primary proof obligation generation)
  updated by: C Analyzer (proof obligation check)
- **<ff>_spo.xml:** Supporting proof obligations for a function.
  created by: C Analyzer (primary proof obligation generation)
  updated by: PyAdvance, C Analyzer (proof obligation check)
- **<ff>_pod.xml:** Dictionary of primary and supporting proof obligation types, using predicate and type/expression indices from <f>_cdict.xml and <f>_prd.xml.
  created by: C Analyzer (primary proof obligation generation)
  updated by: PyAdvance (creation of supporting proof obligations)
- **<ff>_vars.xml:** Dictionary of analysis artifacts referenced in invariants.
  created by: C Analyzer (invariant generation)
  updated by: C Analyzer (invariant generation)
- **<ff>_invs.xml:** Dictionary of invariant values and location invariant table.
  created by: C Analyzer (invariant generation)

updated by: C Analyzer (invariant generation)

4 Dictionary Data Structure Formats

Several file-level and function-level files provide an index representation of commonly used entities such as expressions and locations. Each data item is represented by a list of strings and a list of integers, which can be indices into other dictionaries or immediate values. Each dictionary file generally has multiple tables for related data structures. Below we describe the structure for each of these tables in these files.

4.1 <f>_cdict

The primary dictionary file for a source file is the <f>_cdict.xml file, which includes entries for all entities relevant to the source code in that file, including types and expressions. Below we describe each of the tables included in this file, and for each of the tables a reference to the relevant python files in PyAdvance that provide the data structures for the entities in that table. Basic indexing and access to the _cdict dictionary is provided by `advance/app/CDictionary.py`

4.1.1 Expressions (exp)

PyAdvance reference: `advance/app/CDictionary.py, advance/app/CExp.py`

Constant (CExpConst)

- *tags:* "const"
- *args:* constant-table index

Lval (CExpLval)

- *tags:* "lval"
- *args:* lval-table index

"neg"	arithmetic negation
"bnot"	bitwise complementation
"lnot"	logical not

Table 1: Unary operators

SizeOf (CExpSizeOf)

- *tags:* "sizeof"
- *args:* typ-table index

SizeOfE (CExpSizeOfE)

- *tags:* "sizeofe"
- *args:* exp-table index

SizeOfStr (CExpSizeOfStr)

- *tags:* "sizeofstr"
- *args:* string-table index

AlignOf (CExpAlignOf)

- *tags:* "alignof"
- *args:* typ-table index

AlignOfE (CExpAlignOfE)

- *tags:* "alignofe"
- *args:* exp-table index

UnOp (CExpUnOp)

- *tags:* 1:"unop", 2:operator (see Table 1)
- *args:* 1:exp-table index, 2:typ-table index (result type)

"plusa"	scalar addition
"pluspi"	pointer plus scalar
"indexpi"	pointer plus scalar
"minusa"	scalar subtraction
"minuspi"	pointer minus scalar
"minuspp"	pointer subtraction
"mult"	scalar multiplication
"div"	scalar division
"mod"	scalar modulo
"shiftlt"	bitwise leftshift
"shiftrt"	bitwise rightshift
"lt"	less than
"gt"	greater than
"le"	less than or equal to
"ge"	greater than or equal to
"eq"	equal
"ne"	not equal
"band"	bitwise and
"bxor"	bitwise xor
"bor"	bitwise or
"land"	logical and
"lor"	logical or

Table 2: Binary operators

BinOp (CExpBinOp)

- *tags:* 1:"binop", 2:operator (see Table 2)
- *args:* 1:exp-table index, 2:exp-table index, 3:typ-table index (result type)

Question (CExpQuestion)

- *tags:* "question"
- *args:* 1:exp-table index, 2:exp-table index, 3:exp-table index, 4:typ-table index (result type)

CastE (CExpCastE)

- *tags:* "caste"
- *args:* 1:typ-table index (target type), 2:exp-table index

AddrOf (CExpAddrOf)

- *tags:* "addrof"
- *args:* lval-table index

AddrOfLabel (CExpAddrOfLabel)

- *tags:* "addroflabel"
- *args:* statement index

StartOf (CExpStartOf)

- *tags:* "startof"
- *args:* lval-table index

FnApp (CExpFnApp)

- *tags:* 1:"fnapp", 2:filename
- *args:* 1:linenr, 2:bytenr, 3:exp-table index, 4:opt-exp-list-table index

CnApp (CExpCnApp)

- *tags:* 1:"cnapp", 2:name
- *args:* 1:typ-table index, 2:opt-exp-list-table index

4.1.2 Types (typ)

PyAdvance reference: `advance/app/CDictionary.py, advance/app.CTyp.py`

Note: For all types shown below the listed attributes-table index is optional: it is assumed the type has no associated attributes when this index is absent.

TVoid (CTypVoid)

- *tags:* "tvoid"
- *args:* attributes-table index

TInt (CTypInt)

- *tags:* 1:"tint", 2:integer kind ("ichar", "ischar", "iuchar","ibool", "iint", "iuint", "ishort", "iushort", "ilong", "iulong", "ilonglong", "iulonglong")
- *args:* attributes-table index

TFloat (CTypFloat)

- *tags:* 1:"tfloat", 2:float kind ("float", "fdouble", "flongdouble")
- *args:* attributes-table index

TPtr (CTypPtr)

- *tags:* "tptr"
- *args:* 1:typ-table index (target type), 2:attributes-table index

TArray (CTypArray)

- *tags:* "tarray"
- *args:* 1:typ-table index (element type), 2:opt exp-table index (size), 3: attributes-table index

TFun (CTypFun)

- *tags:* "tfun"
- *args:* 1:typ-table index (return type), 2:opt-fun-args index (argument types), 3:is_varargs, 4:attributes-table index

TNamed (CTypNamed)

- *tags:* 1:"tnamed", 2:name
- *args:* attributes-table index

TComp (CTypComp)

- *tags:* "tcomp"
- *args:* 1:ckey (unique struct identifier), 2:attributes-table index

TEnum (CTypEnum)

- *tags:* 1:"tenum", 2:name
- *args:* attributes-table index

TBuiltinVaargs (CTypBuiltinVaargs)

- *tags:* 1:"tbuiltin-va-list"
- *args:* attributes-table index