KT Advance Reference Manual

Kestrel Technology, LLC

November 24, 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.

can-leave-scope(ptr:exp): The value of the pointer expression can safely leave the scope (it does not point to a memory location that goes out of scope when the block/function is exited).

cast(tfrom:typ,tto:typ,e:exp): The value of the expression can be safely cast from type `tfrom` to type `tto`.

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.

in-scope(ptr:exp): The value of the ptr expression points to a memory region that is in scope.

int-overflow(op:binop,e1:exp,e2:exp,k:ikind): The operation `op` performed on expressions e1 and e2 does not lead to integer overflow, with `k` indicating the width of the integer.

int-underflow(op:binop,e1:exp,e2:exp,k:ikind): The operation `op` performed on expressions e1 and e2 does not lead to integer underflow, with `k` indicating the width of the integer.

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.

pointer-cast(tfrom:typ,tto:typ,ptr:exp): The ptr expression can be safely cast from target type `tfrom` to target type `tto`.

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.

signed-to-unsiged-cast(ifrom:ikind,ito:ikind,scalar:exp): The scalar expression can be safely cast from the signed integer type `ifrom` to the unsigned integer type `ito`, in particular, it should be non-negative.

unsigned-to-signed-cast(ifrom:ikind,ito:ikind,scalar:exp): The scalar expression can be safely cast from the unsigned integer type `ifrom` to the signed integer type `ito`.

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 and has not gone out of scope.

value-constraint(e:exp) Boolean expression that is supposed to evaluate to true.

width-overflow(e:exp,k:ikind): The value of the expression is less than the width of the integer type (in bits).

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 or in proof obligations.
  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)
- **<f>_cgl.xml:** Dictionary of global and field assignments.
  created by: C Analyzer (primary proof obligation generator)
  updated by: C Analyzer (invariant generation)

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 on other functions, global variable condition requests.
  created by: C Analyzer (primary proof obligation generation)
  updated by: C Analyzer (invariant generation, 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)

name	description	AdvancePy reference
attr	program type attributes	app/CAttributes.py
const	program constants	app/CConstExp.py
exp	program (sub)expressions	app/CExp.py
host	program location	app/CLHost.py
lval	program left-hand values	app/CLval.py
offset	location offsets (field, index)	app/COffsetExp.py
string	constant strings	util/StringIndexedTable.py
typ	program types	app/CTyp.py

Table 1: Tables included in the _cdict dictionary

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 data elements contained in these structures. Indices into other tables are referred to by the table name.

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` Table 1 lists all tables included in this dictionary and the PyAdvance.py files that implement the corresponding objects.

4.1.1 Attribute Parameter Table

The attribute parameter table, shown in Table 2 contains attributes that are associated with types in the program.

tags		args				description
1	2	1	2	3	4	
”aint”		value				integer value
”astr”		`string`				string value
”acons”	name					
”asizeof”		`typ`				
”asizeofe”		`attr`				
”asizeofs”		`typsig`				
”aalignof”		`typ`				
”aalignofe”		`attr`				
”aalignofs”		`typsig`				
”aunop”	unop	`attr`				
”abinop”	binop	`attr`	`attr`			
”adot”		`attr`				
”astar”		`attr`				
”aaddrof”		`attr`				
”aindex”		`attr`	`attr`			
”aquestion”		`attr`	`attr`	`attr`		

Table 2: Attribute-table: Tags and arguments for program type attributes

tags			args		description
1	2	3	1	2	
”chr”			ord(chr)		character represented by its ordinal number
”int”	string-rep	ikind			integer constant
”str”		`string`			constant string
”wstr”			*value		wide string (represented by list of integers)
”real”	string-rep	fkind			real number constant

Table 3: Const-table: Tags and arguments for program constants

4.1.2 Constant Table

The constant table, shown in Table 3 contains constants that appear in the program. Integer and real constants are represented as strings as they appear textually in the program.

4.1.3 Expression Table

The expression table contains expressions and subexpressions that represent expressions in the program. Expressions are represented by an indicator tag followed by an optional second tag and one through four table indices or immediate values, as shown in Table 4.

tags		args				description
1	2	1	2	3	4	
”addrof”		lval				address of location
”addroflabel”		stmt index				
”alignof”		typ				alignment of a type
”alignofe”		exp				alignment of type of expr
”binop”	binop	exp	exp	typ		binary expression
”caste”		typ	exp			cast of expression
”cnapp”	name	*exp				custom predicate on expressions
”const”		const				constant
”fnapp”	filename	line-nr	byte-nr	exp	*exp	function application @loc
”lval”		lval				left-hand-side (location)
”question”		exp	exp	exp	typ	? : expression
”sizeof”		typ				size of type
”sizeofe”		exp				size of type of expr
”sizeofstr”		string				size of string
”startof”		lval				same as addrof
”unop”	unop	exp	typ			unary expression

Table 4: Exp-table: Tags and arguments for expressions

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

Table 5: Unary operators

The string values for *unop and* binop are shown in Tables 5 and 6, respectively.

4.1.4 Lval Table

An lval (a value that can be on the left side of an assignment, that is, a program storage location) consists of a base location called a host, and an optional offset, which can be a field or index. Their representations are shown in Table 7.

4.1.5 Type Table

The typ-table contains data types that represent types that appear in the program with elements shown in Table 9. Some elements are optional and can be absent; these are shown between brackets (e.g., `(attrs)`. Some elements are optional and have value -1 if not provided; these are shown in square brackets (e.g., `[exp]` for the array size expression).

”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 6: Binary operators

The type of integer is specified by the string indicated as *ikind*, which can have the following values with the obvious meanings: ”ichar”, ”ischar”, ”iuchar”,”ibool”, ”iint”, ”iuint”, ”ishort”, ”iushort”, ”ilong”, ”iulong”, ”ilonglong”, ”iulonglong”. The type of float is specified by the string indeciate as *fkind* which can have one of the following three values: ”float”, ”fdouble”, ”flongdouble”.

4.1.6 Type Signature Table

4.2 Declarations Tables

Table 10 shows the data representation of the various types of declarations included in the program. The compinfo represents struct and union definitions. The enuminfo represents enum definitions. The init represents initializers for global variables. The typeinfo represents type definitions, associating a name with a type. Finally, the varinfo represents variable definitions. All variables in a program are indexed: within a file each variable has a unique identifier, called `vid`. Global variable identifiers are cross referenced across files via the _gxrefs.xml files. The

	tags		args				description
	1	2	1	2	3	4	
`lval`							
			`host`	`offset`			left-hand side value
`host`							
	”var”	name	id				program variable
	”mem”		`exp`				expression dereference
`offset`							
	”n”						no offset
	”f”	name	key	`offset`			field offset (fieldname,struct identifier)
	”i”		`exp`	`offset`			index offset

Table 7: Lval-table: tags and arguments for program locations

tags		args				description
1	2	1	2	3	4	
”tarray”		`typ`	`[exp]`	`(attrs)`		array (element type, size)
”tbuiltin-va-list”		`(attrs)`				builtin varargs
”tcomp”		key	`(attrs)`			struct/union type identifier
”tenum”	name	`(attrs)`				enum type
”tfloat”	fkind	`(attrs)`				floating point number types
”tfun”		`typ`	`[fun-args]`	is-varargs	`(attrs)`	function (return type, args)
”tint”	ikind	`(attrs)`				integer types
”tnamed”	name	`(attrs)`				type name
”tptr”		`typ`	`(attrs)`			pointer to another type
”tvoid”		`(attrs)`				void type

Table 8: Typ-table: tags and arguments for types

tags		args				description
1	2	1	2	3	4	
”tsarray”	length	`typsig`	`(attrs)`			
”tsptr”		`typsig`	`(attrs)`			
”tscomp”	name	is-struct	`(attrs)`			
”tsfun”		`typsig`	`typsig-list`	is-varargs	`(attrs)`	
”tsenum”	name	`(attrs)`				
”tsbase”		`typsig`				

Table 9: Typsig-table: tags and arguments for type signatures

	tags		args								
	1	2	1	2	3	4	5	6	7	8	9
compinfo											
	name		key	struct?	`attrs`	`*fieldinfo`					
enuminfo											
	name	ekind	`attrs`	`*enumitem`							
enumitem											
	name		`exp`	`loc`							
fieldinfo											
	name		key	`typ`	bitfield	`attrs`	`loc`				
init											
	”single”		`exp`								
	”compound”		`typ`	`*offset-init`							
loc											
			`filename`	byte	line						
offset-init											
			`offset`	`init`							
typeinfo											
	name		`typ`								
varinfo											
	name	storage	vid	`typ`	`attrs`	global?	inlined?	`loc`	address?	param	`init`
	1	2	1	2	3	4	5	6	7	8	9

Table 10: Declarations tables: tags and arguments for program declarations

address? element of the varinfo indicates whether the address of the variable has been taken (that is, if it can be aliased). The param element is a positive integer for formal parameters of functions, indicating its sequence number in the list of parameters (starting at 1), or zero for all other variables.