

DETECT INVALID MEMORY ACCESSES **T1.accessPredictor**

G "Thanks to T1.accessPredictor we could reduce the time spent for finding illegal memory accesses in our engine management projects dramatically. T1.accessPredictor finds these by statically (on the PC) analyzing the binary – without the need of executing the code on the target." - Feedback from a customer who has been using T1.access-Predictor in several projects

CHECK MEMORY ACCESSES BASED ON THE BINARY ONLY

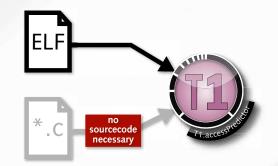
T1.accessPredictor makes it possible to detect access violations before the software runs on the target hardware

Upset by MPU exceptions in the field? Tracking them down while the software executes can be very time consuming and costly. T1.accessPredictor allows you to check for any memory access violations before even flashing the software. Think of it as an "off-line MPU".

AVOID MPU EXCEPTIONS

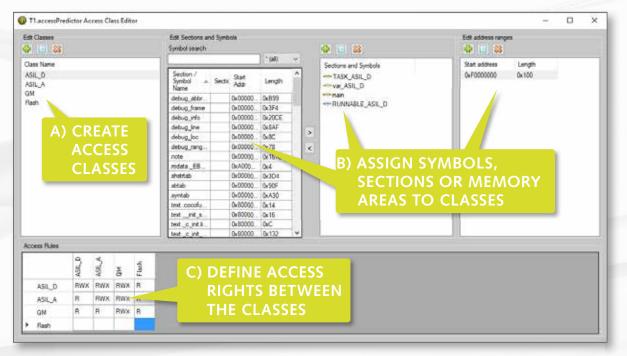
Analyzing the binary rather than the source code has significant advantages

- It is a long way from the source code to the binary and assuming that no additional accesses were injected by the compiler and linker is a critical assumption in a safety-relevant context.
- C source analysis also omits any assembler code.
- What's more today's ECUs incorporate software components from various parties. None of them has a full view on 100% of the sources so a complete analysis is impossible when performing source code analysis.



Using T1.accessPredictor is very simple; there are only a few steps to take

Step 1: Specify the different access classes using the intuitive GUI, add symbols/sections/memory areas and define in which way (Read, Write and eXecute) every class may access the other classes. "Execute" refers to code accesses such as function-calls. In the example, four access-classes were defined: ASIL_A, ASIL_D, QM and Flash.



Step 2: Read in the binary, the ELF file. T1.accessPredictor will disassemble the binary and perform a static analysis based on abstract interpretation. Afterwards T1.accessPredictor presents a "bi-directional" call-tree indicating a) which function calls which other functions and b) by which other functions a function gets called.
Step 3: If necessary, add annotations (manually, generated or measurement-based) to complete the call-tree.
Step 4: Analyze the results! The call-tree indicates access violations with red exclamation marks: for invalid data

System: Core	0	•	Configure	Acce	ss Classes	Show Anno	
Function Names		Data Accesses	Code Accesses	Call	Address	Access Class	
∃-main		St 🚯	N	V	0x800003A6	ASIL_D, Flash	void TASK_ASIL_A()
- TASK_ASIL_A		8	۲	∢_	0x8000036C	ASIL_A, Flash	++var ASIL A;
- RUNNABLE_ASIL_D				1	0x8000000C	ASIL_D, Flash	RUNNABLE ASIL D(); /* Not allowed */
⊨-TASK_ASIL_D		S i	\bigotimes	<	0x8000037A	ASIL_D, Flash	
RUNNA	BLE_ASIL_D			\checkmark	0x8000000C	ASIL_D, Flash	
TASK_QM		(h)		\checkmark	0x80000388	QM, Flash	void TASK QM()
	selected function	Callers for selected	function				++var_QM;
Data Accesses for							++var_ASIL_D; /* Not allowed */
	Data Symbol		Read Write				
Data Address	Data Symbol		Read Write				STM0_CLC = 0x12345678; /* Not allowed
Data Address 0xD0000000	var_ASIL_D		s 🔹 🔶				STM0_CLC = 0x12345678; /* Not allowed
Data Address							STMO_CLC = 0x12345678; /* Not allowed

Step 5 (optional): Export the results for regression tests for subsequent software releases.

GLIWA GmbH embedded systems Pollinger Str. 1 82362 Weilheim i.OB. | Germany

accesses and for invalid code accesses.

gliwa.com

fon +49 - 881 - 13 85 22 - 0 fax +49 - 881 - 13 85 22 - 99 mail: info@gliwa.com