Experiments. We evaluated eVolCheck on a set of industrial benchmarks. Four of them (`VTT_`n) were provided by our industrial partner, the VTT company. The rest is derived from a library of Windows device drivers (`floppy_`n, `kbfiltr_`n, `diskperf_`n). We invented all changes artificially.

Safety of all benchmarks was verified against assertions, either existing in the code or inserted by us into code without assertions. Table I contains results of the experiments. Each row corresponds to a different benchmark, groups of columns represent statistics about the bootstrapping verification and verification of two upgrades, respectively. NoI estimates the size of the original source code as a number of instructions in the `goto-binary` (NoI is an accurate representation of code without definitions, and often represents much higher number of lines of code). IC represents the number of changed instructions between current and the previous version. The overhead introduced by upgrade checking, i.e. the syntactic difference check (Diff) and the interpolants generation (Itp), is also included in the total running time (Total). To show advantages of our upgrade checking approach, for each change we calculated the speedup (Speedup) of the upgrade check versus verification of the changed code from scratch, performed only for the sake of comparison reasons and hidden from the table.

In order to demonstrate different performance of our technique, we chose two different classes of changes for each benchmark. The first class (1st change) represents changes with small impact. As expected, those can be verified with a few local checks. The second one (2nd change) presents upgrades that affect large portion of the code, potentially causing traversal of the complete call tree of the program.

Our experiments demonstrate that for the class of problem with small impact, the upgrade checking approach outperforms the standalone verification (order(s) of magnitude speedup). For the second class of changes, the performance of the upgrade check varies. For some cases, analysis could be done locally and the speedup is still substantial. For cases where the algorithms needed to analyze large portion of the call tree, the performance, as expected, degrades. Note that the bad performance occurs when the change introduces a bug (indicated by '—' in the Itp column; the PBMC formula is satisfiable and interpolants are not generated). In this case, the upgrade check traverses to the root of the call tree, in order to reconstruct a complete error trace. Of course, this can be an easy task when the change is close to the root of the call tree (e.g., in the `floppy_D` benchmark). The results support our initial intuition that upgrade checking works well for incremental changes, which is the most common class in the evolution of systems.

Table I: Experimental evaluation

benchmark		bootstrap		1st change					2nd change				
name	NoI	Total [s]	Itp [s]	IC	Total [s]	Diff [s]	Itp [s]	**Speedup**	IC	Total [s]	Diff [s]	Itp [s]	**Speedup**
`VTT_A`	329	4.889	0.133	2	0.318	0.006	<0.001	**15.6**x	10	15.102	0.006	—	**0.3**x
`VTT_B`	332	23.178	0.003	6	7.793	0.007	0.007	**3.0**x	6	7.805	0.007	0.014	**3.0**x
`VTT_C`	129	0.144	0.001	2	0.017	0.002	<0.001	**8.4**x	1	0.187	0.002	—	**0.8**x
`VTT_D`	247	24.735	0.001	0	0.008	0.008	<0.001	**3098.0**x	2	46.910	0.006	—	**0.8**x
`floppy_A`	292	1.025	0.015	2	0.039	0.009	0.002	**26.1**x	6	0.201	0.009	0.013	**5.0**x
`floppy_B`	294	0.763	0.003	2	0.038	0.009	<0.001	**19.8**x	7	0.046	0.009	0.001	**16.4**x
`floppy_C`	2082	1.280	0.004	2	0.383	0.182	<0.001	**3.4**x	7	0.394	0.183	0.001	**3.2**x
`floppy_D`	2099	60.469	0.257	6	0.374	0.182	<0.001	**161.7**x	23	3.614	0.189	—	**16.8**x
`kbfiltr_A`	529	1.307	0.014	2	0.030	0.011	<0.001	**43.1**x	6	0.111	0.012	0.006	**10.6**x
`kbfiltr_B`	529	1.040	0.001	1	0.052	0.011	0.001	**19.6**x	2	1.835	0.011	—	**0.6**x
`kbfiltr_C`	1010	2.522	0.014	2	0.063	0.021	0.002	**40.2**x	23	0.124	0.021	0.002	**20.3**x
`kbfiltr_D`	1011	3.060	0.009	2	0.061	0.022	<0.001	**50.5**x	7	0.231	0.022	0.003	**7.0**x
`diskperf_A`	486	1.028	0.001	1	0.033	0.008	<0.001	**31.3**x	2	1.751	0.008	—	**0.6**x
`diskperf_B`	492	2.580	0.049	2	0.091	0.009	0.006	**28.3**x	12	2.468	0.009	0.029	**1.1**x
`diskperf_C`	1664	1.126	0.001	1	0.072	0.034	<0.001	**15.6**x	4	0.097	0.034	0.001	**11.5**x
`diskperf_D`	1685	38.609	1.179	1	0.295	0.035	0.016	**130.4**x	2	0.508	0.035	0.020	**75.7**x