AUSPICE: Automatic Safety Property Verification for Unmodified Executables

Erratum and Corrections: 17 May 2016

Jiaqi Tan, Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan

Department of Electrical & Computer Engineering, Carnegie Mellon University
tanjiaqi@cmu.edu,htay@andrew.cmu.edu,rgandhi@ece.cmu.edu,priya@cs.cmu.edu

1 Executive Summary

This document describes corrections to our earlier paper [1]. We update the Safe Function rule (§3.3 and Figure 3 in [1]) to fully specify whole-program safety, and we update the proof sketch for the correctness of the Safe Function rule (§5 in [1]. The Safe Function rule in [1] did not fully cover all required safety-assertion discharges in the CFG of a program. We also point out deficiencies in our earlier correctness proof sketch, and we update the proof sketch for our updated Safe Function rule.

2 The $\mathcal{L}_{LR}$ Program Logic: Updated Safe Function Rule

2.1 Deficiencies in Safe Function rule in [1]

Our Safe Function rule defines what it means for a (source-code level) function in machine-code to possess our safety-properties (of memory-write and control-flow safety, as defined in §2.2 of [1]). Our Safe Function rule (§3.3 and Figure 3 in [1]) has 6 conjuncts which describe the following cases:

1. Address of function being described,
2. Entry CFG node of the function,
3. Exit CFG nodes of the function,
4. Intra-procedural control-flow transfers between basic blocks within the function,
5. Inter-procedural control-flow transfers: function calls where a basic block local to the current function invokes a callee function,
6. Inter-procedural control-flow transfers: function returns from a callee function to a basic block local to the current function.

The Safe Function rule needs to ensure that safety-assertions hold for every possible control-flow transfer (i) within the function, (ii) to callee functions (i.e., function calls), and (iii) from callee functions (i.e., function returns). The Safe Function rule needs to (i) capture every possible control-flow, and (ii) require that the safety-assertions at each successor are discharged by the preconditions of its predecessor in the Control-Flow Graph (CFG) of the program.

HOARE_WITH_ASSERT (§3.3 in [1]) represents a tuple-version of a Hoare triple theorem describing the behavior of a basic block local to the current function, and FUN_SAFE represents a Safe Function theorem describing the safety of a callee function. In the Safe Function rule in [1], conjuncts 4 and 6 did not fully specify all possible control-flow transfers.

The two HOARE_WITH_ASSERT triples in conjunct 4 represent, respectively, a predecessor node and a successor node in a CFG. Then, the successor PC address, *succ* in the successor node (i.e., the second HOARE_WITH_ASSERT triple in conjunct 4), is only existentially quantified in our original Safe Function rule. However, as our HOARE_WITH_ASSERT triples describe single-entry, single-exit basic blocks, to fully specify a CFG node, we need to specify one HOARE_WITH_ASSERT triple for each possible target address that the CFG node can jump to after it has been executed. Hence, existentially quantifying the successor PC (program-counter) address in the second HOARE_WITH_ASSERT triple in conjunct 4 results in under-specification of the safety requirements, as CFG nodes with multiple possible successor PC addresses will have HOARE_WITH_ASSERT triples whose safety-assertions are not specified as needing to be discharged in the Safe Function theorem for the function.

Thus, conjunct 4 of the previous Safe Function rule in [1] allows the safety-assertions in intra-procedural (i.e., receiving control transfers from an intra-procedural basic block) HOARE_WITH_ASSERT triples for CFG nodes with more than one successor PC (i.e., branching blocks) to go undischarged, since the existential quantifier means that for CFG nodes with multiple HOARE_WITH_ASSERT triples, only one of them needs to have its safety-assertions discharged.

In conjunct 6, similarly to conjunct 4, the successor PC of the HOARE_WITH_ASSERT triple for the return-site from a function is existentially quantified. Thus, for callee functions that return to a branching basic block in the current function under analysis, which has more than one possible successor PC, our previous Safe Function rule will under-specify the safety-requirements for the program, and allow the safety-assertions in HOARE_WITH_ASSERT triples with the same address, but different successor PC, to go undischarged (by not requiring them to be discharged in the Safe Function theorem).

Thus, conjunct 6 of the previous Safe Function rule in [1] allows the safety-assertions in inter-procedural (i.e., receiving control transfers from callee functions) HOARE_WITH_ASSERT triples for CFG nodes with more than one successor PC (i.e., branching blocks) to go undischarged, since the existential quantifier means that for CFG nodes with multiple HOARE_WITH_ASSERT triples, only one of them needs to have its safety-assertions discharged.

We note that conjunct 5 does not need to be updated as the predecessor and successor PCs of the calling basic block, and the PC of the Safe Function theorem for the callee function, are all universally quantified, and sufficiently constrained.

2.2 Updated Safe Function rule

Figure 1 describes our updated Safe Function rule, with conjuncts 4 and 6 updated. In both conjuncts 4 and 6, we need to consider all possible successor PCs *succ* for a given `HOARE_WITH_ASSERT` triple of a given node, *node*.

In conjunct 4, we have universally quantified the successor PC of the successor (i.e., second) `HOARE_WITH_ASSERT` triple for intra-procedural control-flow transfers. We now require, for each node *node*, to have all the `HOARE_WITH_ASSERT` triples associated with the node (i.e., all possible CFG successors *succ* for *node*) to be included in the Safe Function rule, and to have their safety-assertions discharged by the pre-conditions of their predecessor nodes' `HOARE_WITH_ASSERT` triples. The successor PCs, *succ*, for a given node *node*, are either local PC addresses of basic blocks in the same function, which are given by the intra-procedural CFG successor map CFG_{succ}, or they can be PC addresses of callee functions, which would be captured by the inter-procedural CFG predecessor map $ICFG_{callpred}$, which returns the callee function addresses called given a basic block address *node* in the current function.

In conjunct 6, we have universally quantified the successor PC of the successor (i.e., second) `HOARE_WITH_ASSERT` triple for intra-procedural control-flow transfers. We now require, for each node *node*, to have all the `HOARE_WITH_ASSERT` triples associated with the node (i.e., all possible CFG successors *succ* for *node*) to be included in the Safe Function rule, and to have their safety-assertions discharged by the pre-conditions of their predecessor nodes' `HOARE_WITH_ASSERT` triples. Similarly to conjunct 4, we require all possible successor CFG PCs *succ* for a given PC *node* to be specified, and they can be intra-procedural PCs (given by the intra-procedural CFG successor map CFG_{succ}), or inter-procedural PCs of callee functions called by the basic block (given by the inter-procedural CFG predecessor map $ICFG_{callpred}$).

3 Proof Sketch of Correctness of Safe Function Rule

3.1 Deficiencies in Correctness Proof Sketch for Safe Function rule in [1]

As explained earlier in §2.1, each CFG node is represented by one single-entry, single-exit Hoare triple theorem for each possible successor PC that control transfers to after the execution of that node. Hence, for a CFG node with N possible successor PCs, it is represented by N Hoare triples (or Safe Function theorems, for inter-procedural targets whose PCs are outside address range of the current function). Our earlier correctness proof sketch did not consider each CFG node as being represented by one or more single-entry, single-exit basic blocks, and omitted considering branching basic blocks. Next, we update our proof sketch of correctness for our updated Safe Function rule.

3.2 Updated Correctness Proof Sketch for Safe Function rule

We reproduce the correctness proof sketch from §5 of [1], and we highlight the updated parts of the correctness proof sketch in blue.

Next, we give a brief, informal argument of the correctness of our proof rule for safe programs. The `FUN_SAFE` theorem (Figure 1) can be proven for a program if and only if safety assertions are specified for every instruction, and if these safety assertions hold before that instruction begins executing (except for the first instruction, which relies on the OS to correctly initialize the processor state for the program). We argue this by Structural Induction on the Control-Flow Graph (CFG) of a program. The CFG of a function in a program consists of nodes and edges. Each node represents either (i) a basic block of a linear sequence of instructions (whose control can transfer out of the basic block only at the end of the basic block) in the function at a given address, or (ii) a callee function that is invoked in the function. Then, an edge in the CFG represents a transfer of control from one node in the CFG (which can be a basic block in the function, or a callee function), to another node in the CFG. A callee function node is associated with a single `FUN_SAFE` theorem which specifies the safety of the callee function. A basic block node (in the function) is associated with one `HOARE_WITH_ASSERT` triple for each possible target that control can transfer to after the basic block has been executed. Thus, for safety to hold at each CFG node, the safety-assertions at all associated `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems for that node must be discharged by the pre-conditions of all the associated `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems for all predecessor nodes of the node.
Base Case. The `MEM_CFI_SAFE` rule (§3.1 in [1]) ensures every instruction's theorem contains our safety assertions (§2.2 in [1]). The `MEM_CFI_SAFE_COMPOSE` rule ensures every basic block's theorem is built up only from single-instruction theorems with added safety assertions. The requirement that post-states of predecessor theorems and pre-states of successor theorems must be equal in `MEM_CFI_SAFE_COMPOSE` ensures every basic block's theorem accumulates the safety assertions for every composed safe instruction theorem. Then, for a program with only a single instruction or basic-block, if the OS correctly initializes the processor state, the safety assertions will hold for the single instruction or single basic block.
Inductive Case. We take the CFG of a function, G, and partition its vertices into a single vertex, g, and all other vertices, G'. By the Inductive Hypothesis, the `FUN_SAFE` theorem holds for G'. Then, consider the edges E connecting G' to g. In the absence of function pointers and unstructured jumps (`longjmp`), the edges E are either (i) intra-procedural control-flow transfers between basic blocks in the function, (ii) function calls from a basic block in the function to a callee function, or (iii) function returns from a callee function to a basic block in the function.

Then, for `FUN_SAFE` to be true, the fourth to sixth conjunct clauses of the `FUN_SAFE` rule must be true. We will see how the 4th to 6th conjuncts cover all possible types of control-flows to and from the node g, so that the pre-conditions of the theorems of all predecessor vertices to g in the CFG discharge

the safety assertions at g, making the safety assertions at g hold, for any type of possible control-flow transfer to g for all `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems associated with g.

In the case of intra-procedural control-flow transfers to g, where g is a basic block within the function being analyzed, g is associated with one `HOARE_WITH_ASSERT` triple for each possible control-transfer target from node g. These targets can either be intra-procedural targets (i.e., the successor node to g is a basic block within the function), or inter-procedural targets (i.e, the successor node to g is a callee function outside the function). The 2nd precedent of the 4th conjunct, $\forall pred \cdot pred \in CFG_{pred}(node)$ ensures all predecessors to g are considered. The 1st disjunct of the 3rd precedent, $\forall succ \cdot succ \in CFG\_succ(node)$ ensures all possible intra-procedural successor nodes to g are considered. The 2nd disjunct of the 3rd precedent, $\forall succ \cdot succ \in ICFG_{callpred}(node)$ ensures all possible inter-procedural successor nodes to g (i.e., callee functions jumped to after g) are considered. Thus, all possible predecessors to the CFG node g are considered, and all possible control-transfer targets from CFG node g are considered.

In the case of inter-procedural control-flow function-calls to g (i.e., g is a callee function, as in the 5th conjunct), `FUN_SAFE` theorems for callee functions g are indexed only by the address of the function, but not by the address of the return-site from the function. Hence, we only need to consider all call-sites of the function. The two precedents in the 5th conjunct (i.e., $\forall node, succ \cdot node \in ICFG_{callsucc}(succ) \Rightarrow succ \in ICFG_{callpred}(node)$) ensure that all call-sites within the function to the callee function are considered.

In the case of inter-procedural control-flow function-returns to g (i.e., g is a basic block within the function, as in the 6th conjunct, and its predecessor is a callee function), the first two precedents of the 6th conjunct, $\forall node, pred \cdot node \in ICFG_{retsucc}(pred) \Rightarrow pred \in ICFG_{retpred}(node)$, ensure that the function from which control is returned to node g is considered. Then, the two disjuncts in the 3rd precedent, $succ \in CFG_{succ}(node) \vee succ \in ICFG_{callpred}(node)$, ensure respectively that the `HOARE_WITH_ASSERT` triples for (i) all possible intra-procedural successor nodes to g, and that (ii) all possible inter-procedural successor nodes to g, are considered.

Thus, our `FUN_SAFE` rule ensures that we have captured all the possible control-flow transfers in a machine-code program. For `FUN_SAFE` to be correct, we require correct CFG predecessor and successor maps, which are straightforward to compute without function pointers and unstructured jumps.

References

1. Tan, J., Tay, H., Gandhi, R., Narasimhan, P.: AUSPICE: Automatic Safety Property Verification for Unmodified Executables. In: VSTTE (2015)

$$
\begin{aligned}
&\vdash \texttt{FUN\_SAFE}(addr, NODES, FUNCS, CFG_{pred}, CFG_{succ}, ICFG_{callpred}, ICFG_{callsucc},\\
&\qquad ICFG_{retpred}, ICFG_{retsucc}, assns_{entry}, postcond_{exit}, prestate, poststate) \Leftrightarrow\\
(\quad &(\forall node \cdot node \in NODES \Rightarrow (\texttt{min}(node, addr) = addr))\\
\wedge\ &(\forall min \cdot min \in NODES \Rightarrow (CFG_{pred}(min) = \emptyset \wedge ICFG_{callpred}(min) = \emptyset \wedge ICFG_{retpred}(min) = \emptyset)\\
&\quad \Rightarrow (\forall node \cdot (node \in NODES \Rightarrow node \neq min) \Rightarrow (\texttt{min}(node, min) = min))\\
&\qquad \Rightarrow \exists pd_1, x, c_1, p_1, q_1 \cdot \texttt{HOARE\_WITH\_ASSERT}(pd_1, assns_{entry}, min, node, x, c_1, p_1, q_1) \wedge\\
&\qquad\quad (prestate = \texttt{aPC}\ min * p_1))\\
\wedge\ &(\forall out \cdot out \in NODES \Rightarrow (CFG_{succ}(out) = \emptyset)\\
&\quad \Rightarrow (\forall funcnode \cdot (funcnode \in FUNCS \Rightarrow out \notin ICFG_{callsucc}(funcnode)))\\
&\qquad \Rightarrow \exists pd_1, assn_1, node, x, c_1, p_1, q_1 \cdot \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, out, node, x, c_1, p_1, q_1) \wedge\\
&\qquad\quad (poststate = q_1) \wedge (pd_1 \Rightarrow postcond_{exit}))\\
\wedge\ &(\forall node, pred, succ \cdot\\
&\qquad\quad (node \in NODES) \Rightarrow\\
&\qquad\quad (pred \in CFG_{pred}(node)) \Rightarrow\\
&\qquad\quad ((succ \in CFG_{succ}(node)) \vee (succ \in ICFG_{callpred}(node))) \Rightarrow\\
&\quad \exists pd_1, assn_1, x, c_1, p, q, pd_2, assn_2, c_2, r \cdot\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, pred, node, x, c_1, p, q) \wedge\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, succ, x, c_2, q, r) \wedge (pd_1 \Rightarrow assn_2))\\
\wedge\ &(\forall node, succ \cdot node \in ICFG_{callsucc}(succ) \Rightarrow succ \in ICFG_{callpred}(node) \Rightarrow\\
&\quad \exists pd_1, assn_1, x, c_1, p, q, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_2, pd_2, r \cdot\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, node, succ, x, c_1, p, q) \wedge\\
&\qquad \texttt{FUN\_SAFE}(succ, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_2, pd_2, q, r) \wedge (pd_1 \Rightarrow assn_2))\\
\wedge\ &(\forall node, pred, succ \cdot\\
&\qquad\quad (node \in ICFG_{retsucc}(pred)) \Rightarrow\\
&\qquad\quad (pred \in ICFG_{retpred}(node)) \Rightarrow\\
&\qquad\quad ((succ \in CFG_{succ}(node)) \vee (succ \in ICFG_{callpred}(node))) \Rightarrow\\
&\quad \exists pd_1, assn_1, x, c_2, p, q, pd_2, assn_2, r, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6 \cdot\\
&\qquad \texttt{FUN\_SAFE}(pred, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_1, pd_1, p, q) \wedge\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, succ, x, c_2, q, r) \wedge (pd_1 \Rightarrow assn_2)) \quad)
\end{aligned}
$$

Fig. 1. Safe Function (FSI) rule: Judgment for Interprocedural Function Safety