Subroutine Verification

Motivation and Problem Statement

The *bytecode verification* is an important part of Java security and is checked by TCK. While modern compilers do not generate subroutines, to pass TCK, we need to support any aspect of the verification including the *subroutine verification*.

We created a modular subroutine inliner, which then reused the data flow analysis for programs without subroutines. Inlining approach allows supporting a bigger subset of type-safe programs [3], for example, the programs generated by bytecode optimizers. While our solution is compatible with specification, it can be easily tuned and extended to different safe program sets.

Definitions

Common Terms

- *Subroutines* are sets of nodes whose stack maps may depend on a calling context. Since the specification disables subroutines to merge their execution paths into a single `ret` instruction we identify subroutines with their `ret` instructions.
- A *subroutine context* is a sequence of `jsr` instructions needed to reach the subroutine.
- *Inlining subroutine* means duplicating subroutine nodes for each calling context. In other words, we need an only call sequence for each subroutine copy.

Special Nodes in a Control Flow Graph

- Graph terminals are *start* and *end*. The *start* node has an out edge to the first basic block of code, and *end* node collects out edges from all `return` instructions.
- jsr_i and sub_i are the nodes corresponding to the *i*-th `jsr` instruction in order of appearance. jsr_i is the node which ends with the correspondent `jsr` call and sub_i is a subroutine entry point.
- ret_j nodes contain the *j*-th `ret` instruction.

Graph Node Marks

Each graph node is either reachable from the first node or not. This is represented by a boolean flag `vf_Node_s.reachable`.

Also, nodes are either a part of subroutine or a top level code:

- A graph node is either an *unknown code*.
- Or a graph node is a *top level code*.
- Or a graph node is a part of a subroutine with a specified ret_j node.

This marking separates subroutines from the rest of the code and provides enough information for a subsequent subroutine inlining.

Finding Subroutines

Here is an algorithm sketch.

- Initially all nodes are marked as not reachable except the *start* node. *start* and *end* nodes are marked as a top level code. We traverse graph recursively following out branches and marking nodes as reachable.
- Any node reachable from the *top level code* is marked as the *top level code*, see `vf_traverse_top_level_code(node)`.
- Nodes reachable from the *unknown code* are marked as the *unknown code*, see `code_t vf_traverse_unknown_code(node)`.
- For jsr_i node we follow a call branch to resolve where the call to a subroutine is returned. The subroutine entry point sub_i starts with the *unknown code* type.
- While traversing the nodes of the *unknown code* type we do the following:
    - Perform a simple data flow analysis for subroutine return addresses. It seems that we get a compatible implementation when identify return addresses with correspondent entry nodes sub_i and merge different entry points into unusable values.
    - We maintain a stack of subroutine entry points sub_i and correspondent jsr_i nodes. If we encounter the same jsr_i again, we report `java.lang.VerifyError: Recursive call to jsr entry`.
- A local variable map allows us to resolve a subroutine entry point when coming to ret_j and a correspondent return address. We have to verify the following at this step:
    - No code loads a return address into local variable.
    - Any return address is used no more than once.
    - Any instruction has a fixed stack depth.
- Before returning from `vf_traverse_unknown_code` we resolve the *unknown code* type of a node in a following way:
    - If any out edges points to ret_j code, we set a code type of the node to ret_j. If there is more than one return from the subroutine, we report `java.lang.VerifyError: Multiple returns to single jsr`.
    - If all node's out edges point to the *top level code*, we mark the node as the *top level code* and return to the previous node.
    - If there are still out edges which point to nodes of the *unknown code* type, we call to `vf_traverse_unknown_code`.

[1] JVM specification.

[2] Leroy, X. Java Bytecode Verification: Algorithms and Formalizations.

[3] Coglio, A. Simple Verification Technique for Complex Java Bytecode Subroutines.