Part III

Bytecode Verification: The Secure JVM

In Part III we extend the trustfully executing JVM of Part II by the verifier and loading components which load and verify bytecode before it may be executed by the trustful VM. We analyze the ambiguities and inconsistencies encountered in the official description in [23], and resolve them by an appropriate new bytecode verifier (Chapter 16).

First we define a diligent VM (Chapter 17) which adds a bytecode verifying submachine to the trustful VM of Part II. We prove the soundness (Theorem 17.1.1) and the completeness (Theorem 17.1.2) of this bytecode verifier. Then we refine the diligent VM to the dynamic VM (Chapter 18) which at run-time loads and references classes when they are needed, before they are passed to the verifier.

The bytecode verifier simulates at link-time, i.e., on type frames instead of value frames, all possible run-time paths through the submitted program, trying to verify certain type conditions which will guarantee the safety of the trustful execution at run-time. We extract these link-time checkable conditions—bytecode type assignments (Chapter 16)—from natural run-time type checks of a defensive VM (Chapter 15). We show that these checks are monotonic, i.e., are preserved when type frames get more specific. In run-time terms, the checking component of the defensive VM can be defined in a modular way by successively refined submachines $check_I$, $check_C$, $check_O$, $check_E$, $check_N$.

In the bytecode verifier the checking machines are coupled to machines $propagateVM_I$ and $propagateVM_E$ (Chapter 17), which propagate checked type frames to possible successor frames (Sect. 16.2), resulting in a link-time simulation of the trustful VM of Part II.

We prove that bytecode type assignments guarantee the desired run-time safety (Theorem 16.4.1), that the compiler of Part II generates code with bytecode type assignments (Theorem 16.5.1), and that our bytecode verifier computes most specific bytecode type assignments (Theorem 17.1.2), thus establishing the Main Theorem of this book. For the completeness proof we refine the compiler to generate code with type information, which is then used in the inductive steps of the proof.