Automata-Based Java Information Flow Analysis (Postprint)

Authors: Wu Zezhi, Chen Xingyuan, Du Xuehui, Yang Zhi

Date: 2018-05-20T00:00:00+00:00

Abstract

Information flow analysis for Java requires modifying the compiler or runtime environment, suffers from poor compatibility with existing systems, and lacks formal analysis and security proofs. First, we propose a Java information flow analysis method based on finite-state automata, which abstracts the taint value space of program variables into the automaton state space and treats Java bytecode instructions as automaton state transition actions. Then, we present information flow security rules for automaton transitions and prove the non-interference security of program execution under these rules. Finally, we implement the prototype system IF-JVM using a method of static taint tracking instruction insertion and dynamic taint tracking and control, which neither requires obtaining Java application source code nor modifies the Java compiler and runtime environment, and is independent of the guest operating system. Experimental results demonstrate that the prototype system can correctly achieve fine-grained information flow tracking and control for Java, with a performance overhead of 53.1%.

Full Text

Preamble

Automata-based Information Flow Analysis for Java

Wu Zezhi[1],[2], Chen Xingyuan[1],[2], Du Xuehui[1], Yang Zhi[1]

[1]College of Cryptogram Engineering, PLA Information Engineering University, Zhengzhou 450001, China
[2]State Key Laboratory of Cryptology, Beijing 100094, China

Abstract: Existing Java-oriented information flow analysis approaches suffer from poor compatibility with current systems due to modifications required in

the compiler or runtime execution environment, while also lacking formal analysis and security proofs. This paper first proposes a formal Java-oriented information flow analysis method based on finite state automata, which abstracts the taint value space of all program variables into an automaton state space and treats Java bytecode instructions as automaton state transition actions. Second, it presents information flow security rules for automaton transitions and proves the noninterference security property under these rules. Finally, it implements a prototype system called IF-JVM using static taint tracking instruction insertion combined with dynamic taint tracking and control. This approach neither requires access to Java application source code nor modifications to the Java compiler or runtime execution environment, and remains independent of the host operating system. Experimental results demonstrate that the prototype system correctly achieves fine-grained information flow tracking and control for Java with a performance overhead of 53.1%.

Keywords: finite state automata; dynamic taint tracking; information flow analysis; noninterference; Java

0 Introduction

Dynamic information flow tracking, also known as dynamic taint tracking or dynamic data flow tracking, refers to monitoring the propagation of data of interest during program execution. The core concept of information flow control mechanisms involves attaching labels to data that propagate throughout the system alongside the data itself, where objects derived from labeled data inherit these labels. These labels are then used to restrict data flow between programs. Confidentiality labels protect sensitive data from unauthorized or malicious access, while integrity labels protect critical information or storage from untrusted or malicious corruption. This technique has been widely applied across various domains of computer system security.

Previous implementations have achieved information flow control at different granularities across multiple layers, including hardware, virtual machine, high-level language, low-level language, operating system, network, and data layers. The work most relevant to this paper involves virtual machine layer and high-level language layer implementations. At the virtual machine layer, TaintDroid implements whole-system dynamic taint tracking within the Dalvik virtual machine, Trishul achieves Java dynamic taint tracking based on the Kaffe VM, and Laminar pioneered an information flow tracking and control system combining virtual machine and operating system approaches. Other related efforts have implemented information flow tracking for JavaScript, Python, PHP, and Java. However, these approaches lack formal security analysis and proofs, and their implementations require modifications to the respective Java virtual machine environments, resulting in poor compatibility. Jif implements static information flow analysis and control based on compiler modifications and provides

noninterference security analysis using a type system, but it primarily operates statically and offers limited support for dynamic information flow tracking and control. This paper synthesizes the advantages of these prior works with the following main contributions:

a) Proposes an automata-based Java information flow analysis method that abstracts the taint value space of all program variables into a state space and treats Java bytecode instructions as automaton state transition actions. This provides a theoretical foundation for Java-oriented information flow control by establishing formal information flow security rules for automaton transitions.

b) Presents a definition of noninterference security based on high-level inputs and low-level outputs, and proves that the automaton state transitions satisfy noninterference security.

c) Implements the prototype system IF-JVM using static taint tracking instruction insertion combined with dynamic taint tracking and control. This approach neither requires Java application source code nor modifications to the Java compiler and interpreter, and remains independent of the host operating system.

---

1 Automata-based Information Flow Analysis

Finite state automata (FSA), hereafter referred to simply as finite automata, are mathematical models for discrete input-output systems widely used in programming language design, implementation, and formal security model verification. An FSA maintains a finite number of states to remember relevant information about past inputs and determines its next state and behavior based on current inputs. A finite automaton can be equivalently viewed as a state transition graph driven by system behaviors (events), where these behaviors include input events, output events, and internal events. Consequently, information flow security policies can be formally described through system state transition rules. In this work, the finite state automaton serves two functions: first, it records the values of all system entities and their security labels along with the sets of all input and output actions; second, it prevents information flow from high-level entities to low-level entities according to security policies.

The automaton is formally defined as follows. Let $V(v, s)$ denote the set of all variables and their security label values within a program. The variable v includes types such as boolean, byte, character, integer, short, long, float, double, object, and arrays. The security label s consists of different basic labels to track multi-source information flows. These security labels form a partially ordered lattice $(L, \sqsubseteq, \wedge, T, \perp)$ where the meet operation $\wedge$ corresponds to set union $\cup$. For any security labels x and y: (1) $x \wedge x = x$ (idempotence); (2) $x \wedge y = y \wedge x$ (commutativity); (3) $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (associativity). The

top element is the empty set $\emptyset$, denoted as T, where $T \wedge x = x$ for all $x \in L$. The bottom element is the universal set U, denoted as $\perp$, where $\perp \wedge x = \perp$ for all $x \in L$.

Let a denote a system action that can cause automaton state transitions. System actions consist of different types of instructions. As shown in [Figure 1: see original paper], we categorize system actions into nine primary types: arithmetic operations, initialization, return operations, load operations, store operations, constant operations, jump operations, and stack operations. For arithmetic operations (containing 14 subtypes), consider the add instruction $a = a + b$: it performs addition on variable values v and updates a's security label to the meet of a and b' s labels. Initialization actions (3 subtypes) include operations like `newarray`, which creates arrays of specified lengths along with corresponding security labels. Return operations, such as `dreturn`, return both the stack-top value and its security label. Load and store operations handle both operands and their security labels. Constant operations assign security label 0 to constants. Jump operations and other control flow instructions follow similar principles. Instructions that do not alter automaton states, such as unconditional jumps (`goto`) and void returns (`return`), are not considered system actions.

Let A denote the set of action sequences composed of sequentially executed actions. Let pc represent the security label of the program counter, and $stack$ represent the security label of data stored in the virtual machine stack. The automaton state is defined as $Q = V \times S$, and state transitions are described by the transition function $\delta : Q \times A \to Q$. Let q_0 denote the initial automaton state. Let P represent the Java program (comprising data and instructions), I the program input set (including keyboard input, file reads, database queries, or network receives), and $I_H \subseteq I$ the set of high-level inputs such as passwords or private files. Let O denote the program output set (including screen display, file writes, database writes, or network sends), and $O_H \subseteq O$ the set of high-level outputs containing variables influenced by high-level inputs. Let κ represent the declassification capability of output variables—for instance, if variable v is influenced by high-level input but the output is internal, the system remains secure for usability and flexibility.

The automaton system can be fully described as (Q, Σ, δ, q_0). The state transition rules are defined as follows:

[The specific transition rules would be listed here, preserving the mathematical notation exactly as in the original.]

To facilitate understanding, Table 1 illustrates a sample program and its automaton state transitions. For the first statement, applying the allocation rule adds variable a and its security label to the automaton state. The input rule then sets a' s value and security label based on the input. Similar operations apply to the second statement. The third statement applies the constant rule, setting the constant' s security label to $\emptyset$. The fourth statement uses the invocation rule to pass actual parameter values and labels to formal parameters. The

eighth statement applies the arithmetic rule, updating c' s label to the meet of c and d' s labels. The ninth statement applies the return rule, setting the return value and its label and removing local parameters from the automaton state. The fifth statement' s output rule shows that since *str*' s security label is empty, the output is safe. The sixth statement' s jump rule reveals that since b' s security label is non-empty, the jump is unsafe, causing the automaton to reject the statement and prevent execution of the seventh statement.

---

2 Security Proof

The security proof employs noninterference analysis, an abstraction that captures security essence. Intuitively, if protected and unprotected data do not interfere with each other, the system is considered secure. In this context, noninterference means that high-level program inputs do not affect public outputs, thereby preventing information flow from high-level inputs to public outputs. Noninterference security is formally defined as: for any two different types of high-level inputs, the system' s public outputs are equivalent.

To prove the system satisfies noninterference security, we first define and prove the monotonicity of security label propagation. Monotonicity is defined as: (1) an automaton state transition represented by function f satisfies $f(x \wedge y) \sqsubseteq f(x) \wedge f(y)$ for security labels x and y in the lattice. This is equivalent to: (2) $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. We prove these definitions equivalent:

First, monotonicity (2) implies (1): Since $x \wedge y$ is the greatest lower bound of x and y, we have $x \wedge y \sqsubseteq x$ and $x \wedge y \sqsubseteq y$. By monotonicity (2), $f(x \wedge y) \sqsubseteq f(x)$ and $f(x \wedge y) \sqsubseteq f(y)$. As $f(x \wedge y)$ is the greatest lower bound of $f(x)$ and $f(y)$, monotonicity (1) holds.

Second, monotonicity (1) implies (2): Assuming $x \sqsubseteq y$, we have $x \wedge y = x$. By monotonicity (1), $f(x \wedge y) \sqsubseteq f(x) \wedge f(y)$. Since $x \wedge y = x$, we have $f(x) \sqsubseteq f(x) \wedge f(y)$. As $f(x) \wedge f(y)$ is the greatest lower bound, $f(x) \wedge f(y) \sqsubseteq f(y)$. Thus $f(x) \sqsubseteq f(x) \wedge f(y) \sqsubseteq f(y)$, yielding $f(x) \sqsubseteq f(y)$, which proves monotonicity (2).

For any security labels X and Y, we have $X \subseteq X \cup Y$. Based on this, the system' s noninterference security is proven as follows:

Consider the first case: Assume the automaton initial state q_0 and a state transition sequence decomposed into n actions $a_1, a_2, \ldots, a_n$, where a_n is a public output action. By the output rule, the security label for this public output action is empty. Let a_1 be a high-level input action, transitioning the state from q_0 to q_1. By monotonicity, the security label for the public output action remains empty. By induction, all preceding actions maintain empty security labels for the public output. Therefore, the high-level input action does not affect the public output action. For any different types of high-level inputs, the public output' s security label remains empty, proving noninterference.

The above noninterference considers only public outputs (typically ensuring public output noninterference guarantees system security). For internal outputs, the system satisfies conditional noninterference: if the high-level input and declassification capability satisfy a partial order relation, internal outputs are equivalent. The proof follows similar reasoning, showing that high-level inputs satisfying the partial order relation do not affect internal outputs.

---

3 System Implementation and Testing

3.1 System Implementation

The prototype system IF-JVM employs dynamic taint tracking technology, combining static taint tracking instruction insertion with dynamic taint tracking and control to achieve fine-grained information flow tracking and control for the Java virtual machine. This approach neither requires Java application source code nor modifies the Java compiler or interpreter, and remains independent of the host operating system.

To implement fine-grained data labeling, each program variable requires a corresponding shadow variable for taint storage. For clarity, Table 2 provides source-code-level examples of taint tracking code insertion. The taint variable type, denoted as `taint`, can be configured based on tracking granularity and requirements; this paper defines `taint` as a 32-bit unsigned integer. Program variables include primitive types (e.g., `int a`, `double b`) and reference types. Primitive types receive shadow variables like `a_tag` and `b_tag`. Reference types include instantiated objects (with shadow variables like `class_tag`) and arrays. For primitive type arrays such as `byte[] array`, we create shadow arrays `taint[] array_tag` where each array element maps to a corresponding taint value. Object type arrays store taint values within their respective objects.

The Java virtual machine stack comprises two parts: the operand stack and local variable area, collectively forming a method frame. For method invocation `plus(c, d)`, a new method frame is created with parameters and their taints passed as `plus(c, c_tag, d, d_tag)`. Upon method return, the value and its taint are stored in a container and returned using `return taintint.valueof(c, c_tag)`, after which the method frame is destroyed.

To achieve fine-grained information flow tracking, taint propagation must be implemented for every instruction that can propagate taint. Representative instructions are shown in Table 3 . For fine-grained information flow control, data taint marking and checking are performed at critical program points. Users can predefine taint types for method return values (e.g., keyboard input) and taint checks for method parameters (e.g., SQL statement execution) via configuration files. Additionally, users can employ `SetTaint(val, taint)` and `GetTaint(val)` to set and retrieve taint labels for any variable.

3.2 System Testing

The test environment consists of Ubuntu 12.04 OS, 2.27 GHz CPU, 4 GB RAM, and OpenJDK "IcedTea" JVM 1.8.0, using the Scalabench benchmarking suite.

First, we measured execution time across sub-test suites, running each 10 times and recording averages (detailed sub-test information available at http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite/). Results in Figure 2 [Figure 2: see original paper] show IF-JVM incurs an average 53.1% runtime overhead compared to standard JVM, due to executing additional statically inserted taint propagation instructions alongside the original program instructions. The `scaladoc` sub-test exhibits maximum overhead because its extensive array and string manipulation instructions generate numerous taint propagation operations.

Second, we measured maximum heap memory usage during execution, shown in Figure 3 [Figure 3: see original paper]. IF-JVM consumes 323.1% more heap memory on average than JVM, as the statically inserted taint data structures require additional memory beyond the program' s original data structures.

---

4 Conclusion

Addressing the limitations of existing Java information flow analysis approaches —namely poor compatibility due to compiler or runtime environment modifications and lack of formal analysis and security proofs—this paper proposes an automata-based Java information flow analysis method and proves its noninterference security property. The prototype system IF-JVM, implemented via static taint tracking instruction insertion and dynamic taint tracking and control, correctly achieves fine-grained information flow tracking and control for the Java virtual machine, as demonstrated by experimental results. Future work will focus on optimizing the instrumentation system to make inserted taint instructions more concise and efficient.

References

[1] Wu Z Z, Chen X Y, Yang Z, et al. Research progress on information flow control [J]. Journal of Software, 2017, 28(1): 135-159.

[2] Crandall J R, Chong F T. Minos: Control data attack prevention orthogonal to memory model [C]// Proc of the 37th International Symp. on Microarchitecture. LA: IEEE Press, 2004: 221-232.

[3] Kemerlis V P, Portokalidis G, Jee K, Keromytis A D. Libdft: Practical dynamic data flow tracking for commodity systems [J]. ACM SIGPLAN Notices, 2012, 47(7): 121-132.

[4] Krohn M, Yip A, Brodsky M, et al. Information flow control for standard OS abstractions [C]// Proc of ACM SIGOPS Operating Systems Review. New York: ACM Press, 2007: 321-334.

[5] Schultz D, Liskov B. IFDB: decentralized information flow control for databases [C]// Proc of the 8th ACM European Conference on Computer Systems. New York: ACM Press, 2013: 43-56.

[6] Enck W, Gilbert P, Chun B G, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones [C]// Proc of OSDI. Berkeley: USENIX Association, 2010: 255-270.

[7] Nair S K, Simpson P N D, Crispo B, et al. A virtual machine based information flow control system for policy enforcement [J]. Electronic Notes in Theoretical Computer Science, 2008, 197(1): 3-16.

[8] Roy I, Porter D E, Bond M D, et al. Laminar: practical fine-grained decentralized information flow control [J]. ACM SIGPLAN Notices—PLDI, 2009, 44(6): 63-74.

[9] Haldar V, Chandra D, Franz M. Dynamic taint propagation for Java [C]// Proc of Computer Security Applications Conference. 2006: 301-311.

[10] Matej V, Binder W, Hauswirth M. ShadowData: shadowing heap objects in Java [C]// Proc of ACM Sigplan-Sigsoft Workshop on Program Analysis for Software Tools and Engineering. New York: ACM Press, 2013: 17-24.

[11] Chandra D, Franz M. Fine-Grained information flow analysis and enforcement in a Java virtual machine [C]// Proc of the 23rd Annual Computer Security Applications Conference. New York: IEEE Press, 2007: 363-372.

[12] Manivannan K, Wimmer C, Franz M. Decentralized information flow control on a bare-metal JVM [C]// Proc. of the 6th Annual Workshop on Cyber Security and Information Intelligence Research. New York: ACM Press, 2010: 64-74.

[13] Myers A C, Liskov B. Protecting privacy using the decentralized label model [J]. ACM Trans on Software Engineering and Methodology, 2000, 9(4): 410-442.

[14] Myers A C. JFlow: practical mostly-static information flow control [C]// Proc of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1999: 228-241.

[15] Blackburn S M, Garner R, Hoffmann C, et al. The dacapo benchmarks: Java benchmarking development and analysis [C]// Proc of OOPSLA. New York: ACM Press, 2006: 169-190.

Note: Figure translations are in progress. See original paper for figures.