Towards Security-aware Program Visualization for Analyzing In-lined Reference Monitors*

Aditi Patwardhan, Kevin W. Hamlen, and Kendra Cooper
Department of Computer Science
The University of Texas at Dallas
Email: aditi.patwardhan@student.utdallas.edu, {hamlen,kcooper}@utdallas.edu

***Abstract*—In-lined Reference Monitoring frameworks are an emerging technology for enforcing security policies over untrusted, mobile, binary code. However, formulating correct policy specifications for such frameworks to enforce remains a daunting undertaking with few supporting tools. A visualization approach is proposed to aid in this task; preliminary results are presented in this short paper. In contrast to existing approaches, which typically involve tedious and error-prone manual inspection of complex binary code, the proposed framework provides automatically generated, security-aware visual models that follow the UML specification. This facilitates formulation and testing of prototype security policy specifications in a faster and more reliable manner than is possible with existing manual approaches.**

I. Introduction

Software security is becoming increasingly important with the growth of the Internet and mobile code technologies like Java. Mobile code technologies generate software components for environments in which *code-consumers* receive code from separate *code-producers*. These components are mainly distributed as binary executable files that are downloaded from web pages or as email attachments. In many realistic settings, not all code-producers are fully trusted; for example, web pages may be served from untrusted servers or emails may arrive from untrusted senders. Security is an obvious concern in such an environment. Violations can range from information leakage to access control violations and data corruption.

Secure mobile code environments constrain the behavior of untrusted code by enforcing liveness or safety properties. Many practical policies can be formulated as safety policies. A classic example is the confidentiality policy that prohibits network-send operations after the process has read from a confidential file. This prevents the untrusted process from divulging the file's content over the network. These *history-based* (i.e., stateful) safety policies can be encoded formally as *security automata* [1]. A security automaton is a finite state automaton that accepts policy-permitted sequences of security-relevant events. Bisimulation of the security automaton with the untrusted program is used as a mechanism to enforce the underlying security policies. When an impending violation is detected by the automaton, the offending process is terminated.

In-lined reference monitors (IRM's) [1] have been proposed as a mechanism to insert security automata into untrusted binary code by modifying it (e.g., [2]–[4]) prior to execution. This results in *self-monitoring code* that is guaranteed to self-terminate before policy violations occur. IRM's provide a powerful means to enforce application-specific security policies, but identifying and defining a good security policy, particularly at the bytecode level, is challenging. For example, to prohibit network-send operations, one must be able to rigorously define what constitutes a network-send operation. This may involve considering hundreds of primitive instructions that constitute potentially security-relevant operations, each of which must be identified in a complete, correct specification of the policy. In general, much manual analysis and inspection of malicious and non-malicious binary code may be required in order to formulate a policy that prohibits all undesired program behaviors without curtailing desired behaviors.

A visualization approach, with tool support, is needed to aid the analysis of the untrusted binary code and to facilitate faster and more reliable discovery and prototyping of application-specific security policies. In this short paper, we present a visualization framework that generates security-aware UML-based visual models for the low-level Java bytecode; preliminary results are reported. The framework supports static and dynamic models. The static visual model represents the underlying class structures; the dynamic visual model represents the possible execution sequences (control-flows) in the application. Each execution sequence is mapped to the set of corresponding state transitions of the security automaton (given by the security policy); a user-defined *color code* is provided to visually map the control-flow blocks to the corresponding security automaton states and indicate possible security violations. The approach and tool support have been validated using a classic confidentiality policy and two test applications that violate this policy by leaking information over the network.

The remainder of this paper is organized as follows. Section II discusses the existing tool support for analysis of bytecode and other related work. Section III describes our visualization framework proposed for the security analysis, and the prototype proof of concept that we have developed. Section IV summarizes and proposes future work.

II. Related Work

Practical IRM systems constitute a growing body of past work (c.f., [5]). The Java-MOP system [3] combines runtime monitoring with aspect-oriented programming. It specifies the

*Supported by AFOSR YIP award FA9550-08-1-0044

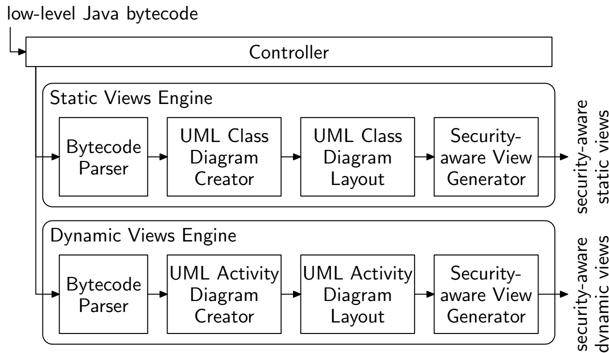

Fig. 1. A security-aware visualization framework

desired security properties, along with the code to execute if a security violation is detected. The specification is then translated to AspectJ code and integrated into the application program using an aspect weaver. The SPoX (Security Policy XML) system [4] provides a purely declarative policy specification language in which security-relevant events are designated via AspectJ pointcuts, and policies over these events are specified as security state transitions. We developed our visualization tool based in part on bytecode analysis libraries provided by the SPoX toolkit.

Traditional text-based, code-level visualization is supported by many established tools including decompilers (e.g., [6], [7]), debuggers, and various low-level libraries for static code analysis. Eclipse [8] provides an integrated debugger for Java source code that allows execution of the source code interactively by stepping through each line of code. Some open source byte code debuggers are also available for finding the trace of execution at a binary code level. Reverse engineering tools (e.g., [9], [10]) and context-independent analyses (e.g., [11]) augment such analyses with support for abstraction visualization and binary format discovery.

However, all of these tools are general purpose; they do not provide specialized support for security. For example, policy-violating control-flows are not automatically identified for the user; they must be manually discovered and extracted. The modeling tools (e.g., [9]) use UML as the modeling notation, which is a powerful and established standard for the graphical modeling of object oriented software design and analysis. We take the approach of catering to security-specific requirements while utilizing the powerful features of a visual modeling notation like UML and using the concept of abstraction for a better representation of the underlying bytecode.

The BCEL API (Byte Code Engineering Library) [12] provides a programmatic foundation for analyzing Java bytecode. Our visualization framework uses this API to extract low-level Java bytecode information.

Graphical models of low-level code provide easier, faster, and more reliable analysis of an application's structure and its possible control-flows than their text-only counterparts. We therefore adopt a graphical approach. Desirable quality attributes and functional requirements for general-purpose code visualization tools have been well-studied [13], but there has been no similar study of security-aware tool functionality.

III. Security-aware Visualization Framework

Figure 1 depicts the structure of our visualization framework, which transforms low-level bytecode into UML-based visual models [14], [15]. It is composed of a controller and separate view engines for the generation of static and dynamic visual models. The *static views engine* generates the view for the static structure of the application modeled as a UML class diagram. The *dynamic views engine* generates the dynamic views for the detailed control-flow diagrams for each class method within the application code, modeled as a UML activity diagram. These UML specifications are used to provide the security-aware views of the application. We choose the Unified Modeling Language (UML) specification for our visual model, as it has become the de facto standard for visual modeling of software applications. We represent the class structures and their relationships using *UML class diagrams*, and the control-flows using *UML activity diagrams*.

The UML class diagrams can be used to compare the untrusted bytecode with the self-monitoring, rewritten code obtained from an IRM framework. The UML activity diagrams can be used to map the prototype security policy to the underlying control structure and to visually identify the security events in the control-flows.

Our visualization framework is built atop the Eclipse plug-in architecture [8]. It can be launched within the Eclipse environment or used as a standalone rich client desktop application. The plug-in architecture allows for easy extensibility to add further custom diagrams related to the security specifications. Further, it does not require any special setup; the only main requirement is a standard Java runtime environment. The prototype tool developed is available at the URL http://utdallas.edu/~aap085000/VisualizationTool.

A. Security-Aware Static View

To effectively prototype and analyze real IRM's and the policies they enforce, it is important to be able to easily visualize and compare the class structure of original and IRM-modified Java bytecode applications. For example, most practical policies constrain usage of certain security-relevant system classes by untrusted applications. The IRM must therefore track the security state of these security-relevant objects at runtime to enforce the policy. The IRM typically accomplishes this by injecting new wrapper classes that inherit from and extend the system classes with extra security state fields maintained by the IRM [3], [4]. Thus, visualizing the class structure of original and IRM-modified applications reveals much about the potential effects of the policy-enforcement upon the untrusted application, including undesired side-effects and potential runtime overhead.

The UML class diagrams [14] represent the static structure of the original and/or IRM-modified system as a structure of classes and relationships between them. They are generated via four pipelined components: a *bytecode parser*, *UML class diagram creator*, *UML class diagram layout*, and *security-aware view generator*.

The bytecode parser identifies the classes, their data attributes and methods, the visibility options of the data attributes and methods, and the relationships between the classes. The UML relationships supported include generalization (based on class inheritance) and association (based on class object reference).

The diagram creator generates the entities of the UML class diagram that represent the information extracted by the parser. Classes are mapped to UML class elements, and class relationships are mapped as generalization or association relationships of the UML class diagram. Supported UML class diagram elements include classes, data attributes, operations, visibility attributes, generalization relationships, and association relationships. We construct the class diagram with inheritance relationships up to one level into the system libraries. Since the class hierarchy is a tree rooted at `java.lang.Object`, this yields a strong inheritance-based structure.

These UML diagram entities are then input as graph elements to the layout algorithm to generate the visual diagram. We use an inheritance-based structure similar to the one in [16] for the automatic layout generation of the class diagram with minimal edge cross-overs. The visualizer allows users to manually adjust the generated layout via a select, drag, and drop functionality for the class elements of the diagram.

The framework can additionally render a visual comparison between the original and the rewritten, IRM-modified application bytecode. This is extremely useful for analyzing changes in the static structure that result from enforcement of a given policy by an IRM. Separate UML class diagrams are generated for the original and IRM-modified application bytecode. Visual color-coded cues are provided to the changes made to the original bytecode during the rewriting process for enforcing the policy. Currently our prototype supports this at the class level granularity; any new classes added to the bytecode by the IRM are highlighted in the class diagram.

The framework's static model visualization has been validated using a test application that divulges a confidential file by sending its content over the network. We used the SPoX IRM system [4] to enforce a policy that prohibits write-access to the Java `Socket` library once a confidential file has been accessed. The class structures of the original, unsafe application bytecode and the SPoX-modified bytecode were then compared using the visualization framework. The new security class injected by the IRM was identified and highlighted in the visual model. Due to space constraints, the screen shots for the before and after class diagrams are not included here; they are available in [17].

B. Security-Aware Dynamic View

UML activity diagrams are used to illustrate the control-flows in a system [14], [15]. Activities typically consist of a network of nodes and edges that represent the flow within the activity; we therefore use them to represent the control-flows. The dynamic views engine consists of four pipelined components: a *bytecode parser*, a *UML activity diagram creator*, a *UML activity diagram layout* and a *security-aware view generator*.

The bytecode parser extracts instruction sequences that are partitioned into *basic blocks*—subsequences of consecutive instructions for which all control-flows (other than exception flows) enter at the beginning and leave at the end without intermediate branching. (Exceptions may result in premature exit from a basic block, and therefore receive special treatment described below.)

The UML activity diagram creator maps these basic blocks as the *call action nodes* of the activity diagrams. Call action nodes define the units of work that are atomic within the activity [15], and are therefore well-suited to basic blocks. Each basic block that ends in a conditional branch introduces a decision node to the diagram. Basic blocks that could throw exceptions caught by a local handler are visualized via control-flow edges from the basic block to the entry-point of the local exception handler. For visual clarity, the control-flow edges to exception handler blocks are shown as dashed arrows to differentiate from the normal control-flow edges.

The UML activity diagram constructs include call action nodes, control-flow edges, and *control nodes*, including an *initial node*, *final nodes*, and *decision nodes*. We use a dataflow analysis technique to identify all possible control-flows in the control structure of the activity diagram by traversing them statically. The UML diagram layout generates a flowchart-like layout that contains call action nodes ordered by underlying bytecode offsets, and the control-flows between them.

The security-aware view generator further provides security-aware views of the activity diagram generated. It takes an input security policy from the user and maps the policy to the control-flows depicted by the activity diagram. We detect all possible policy violations in the control-flows by computing a function $f : Q \to 2^S$ that maps each node $q \in Q$ in the control-flow graph to a conservative approximation of the set of security automaton states $s \in S$ that the node could assume at runtime. The computation involves obtaining the least fixed point of the functional $F : (Q \times S) \to (Q \times S)$ defined by

$$F(g) = g \cup \{(q_0, s_0)\} \cup \{(q', \delta(q, s)) \mid (q, q') \in E, s \in g(q)\}$$

where q_0 and s_0 are the start states of the control-flow graph and security automaton (respectively), $E \subseteq Q \times Q$ is the transition relation for the control flow graph, and $\delta : (Q \times S) \rightharpoonup S$ is the transition function for the security automaton, which defines how each basic block modifies the security state when executed. Our current implementation is intra-procedural; an inter-procedural extension is left for future work.

The visualizer then identifies sites of potential policy violations by identifying the control-flow graph nodes $q \in Q$ for which there exists a security state $s \in (\mathit{fix}(F))(q)$ such that $(q, s) \notin \delta^{\leftarrow}$. These are the states for which the security automaton has no transition, and that therefore might exhibit a policy violation at runtime. These nodes q are therefore the sites where an IRM will typically in-line runtime security checks to detect and prevent potential violations. The visualizer renders these nodes in a unique, user-specified color to

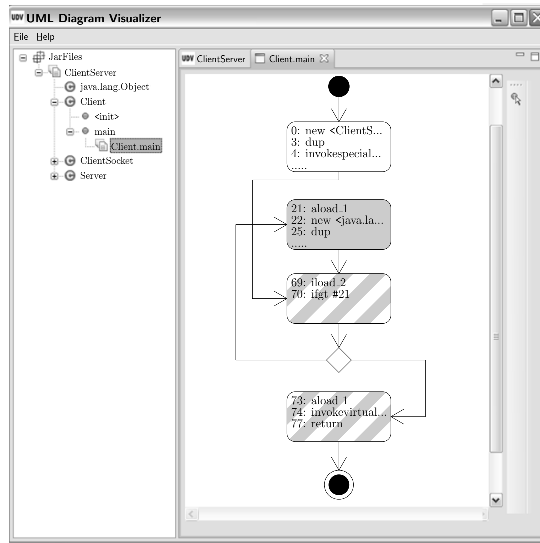

Fig. 2. A UML activity diagram with control-flow nodes color-coded by possible security state

bring them to the attention of the user.

We take a conservative approach in which the detection of policy violations may include some false positives. For example, a potential violation might be identified within an execution branch that is never actually traversed at runtime due to the value of some trusted input variable. However, false negatives will not occur; all potential violations are conservatively identified.

The framework's dynamic model visualization has been validated using a test client-server application (Figure 2). The screen shot illustrates the main method of the client, which contains a loop with a send and a read operation. The security policy prohibits sends after reads. The visualizer detects that this could result in a policy violation for control-flows that exhibit two or more iterations of the loop body. The generated high-level view of the control structure maps the possible security states of the automaton that each basic block could enter, thereby identifying those where a security violation could result. In this case the white node indicates that the security automaton is in an initial state, the striped nodes indicate that the security automaton may be in various different policy-adherent states on various different runs, and the filled node indicates a possible policy violation.

IV. Conclusions and Future Work

We have introduced a security-aware, bytecode visualization framework that facilitates fast and easy prototyping and analysis of IRM security policies and their implementations. This is the first approach to address this concern. Experiments show that the framework, implemented as a prototype tool, can generate visual diagrams and identify code points where possible security violations could occur at runtime. Without an automated tool, such analyses are extremely difficult and time-consuming even for experts, since they involve manually understanding an application's binary structure, its possible control-flows, and its potentially security-relevant operations. Rapid development and prototyping of candidate security policies is therefore typically impractical without such a tool.

A number of limitations have been identified in the visualization framework. With respect to the UML diagrams created, for example, the framework does not support reverse engineering to the complete, standard definitions of UML class and activity diagrams. The current subset has been adequate for the example applications used in the validation, but may need to be extended in the future. In addition, the security-aware color coding scheme used has not been rigorously defined as a UML extension, such as a profile or a stereotype. The graph based algorithms to create the diagrams also need to be systematically defined and analyzed.

The approach taken to identify the possible security violations may include false positives, wherein some unreachable control-flows are identified as possible sources of policy violations. However, the approach does not suffer from false negatives; it conservatively identifies all violations.

The framework can be easily extended to provide further support for visualization with respect to security, including extended debugging functionality and support for visual modeling. These are avenues we intend to explore in future work.

References

[1] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[2] Ú. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proc. New Security Paradigms Workshop*, September 1999, pp. 87–95.

[3] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," *Lecture Notes in Computer Science*, vol. 3440, pp. 546–550, 2005.

[4] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *Proc. ACM Workshop on Programming Languages and Analysis for Security*, 2008.

[5] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of non-safety policies," *ACM Transactions on Information and System Security*, vol. 12, no. 3, January 2009.

[6] P. Kouznetsov, "JAD: The fast Java decompiler," http://www.kpdus.com/jad.html.

[7] Ahpah Software, Inc., "The SourceAgain decompiler," http://www.ahpah.com/products.html.

[8] "The Eclipse platform," http://www.eclipse.org.

[9] International Business Machines, "IBM software architect," http://www-01.ibm.com/software/awdtools/swarchitect.

[10] H. M. Kienle and H. A. Müller, "Rigi: An environment for software reverse engineering, exploration, visualization, and redocumentation," *Science of Computer Programming*, vol. 75, no. 4, pp. 247–263, 2010.

[11] G. Conti, E. Dean, M. Sinda, and B. Sangster, "Visual reverse engineering of binary and data files," *Lecture Notes in Computer Science*, vol. 5210, pp. 1–17, 2008.

[12] "Byte code engineering library," http://jakarta.apache.org/bcel/.

[13] H. M. Kienle and H. A. Müller, "Requirements of software visualization tools: A literature survey," in *Proc. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, June 2007.

[14] B. Gruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, 2nd ed. Prentice Hall, 2004.

[15] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-oriented Analysis and Design*, 2nd ed. Addison-Wesley, 2005.

[16] J. Seemann, "Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams," *Lecture Notes in Computer Science*, vol. 1353, pp. 415–424, 1997.

[17] A. A. Patwardhan, "Security-aware program visualization for analyzing in-lined reference monitors," Master's thesis, University of Texas at Dallas, June 2010.