Design of Software Security Verification with Formal Method Tools

Seung-Ju Jang[†], *Jungwoo Ryoo*[††] *and ChangYeol Lee*[†]

[†]University of Dongeui, [††]University of Penn State

Summary

Formal methods ensure the stability and reliability of soft-ware systems by using mathematical principles and proving conformance to a given set of requirements. The stable and reliable operation of software is especially important for system applications dealing with security. Although very effective in identifying a non-conformance in security requirements, formal methods typically involve a steep learning curve before full adoption. Automated tools can be used to alleviate difficulties associated with formal methods. An observation is made that the existing attempts to apply formal methods to check conformance to security requirements, have not efficiently taken advantage of such tools. Therefore, this paper proposes a novel methodology to leverage well-known formal method tools to verify how closely a security software product satisfies its requirements. More specifically, this paper formally verifies an Access Control System (ACS) using RoZ and Z/EVES, two of the many verification tools available for ensuring the integrity of software applications. For this, a UML model of ACS with Z annotations is first created. Next, the model is transformed into a Z specification which is, in turn, verified by the Z/EVES prover. Using this process, one can also find security vulnerabilities created during a development process. Index Terms—formal methods, formal specification, formal verification, software security verification, RoZ, Z/EVES

Key words:

Software Security, Verification, Formal Method Tools, RoZ.

Introduction

Formal methods can provide mathematical specifications for security software. Their inherent preciseness (1) re-moves ambiguity (2) allows one to examine inconsistencies, and (3) guarantees the stability of a system to be implemented. A set of tools can improve the efficiency of formal specifications efforts by automating certain aspects of them such as analyzing the semantics. When not followed by a verification process, a specification alone is insufficient to ensure the run-time stability of a software system. Tools supporting formal verification include SPIN [1], SLAM [2], Symbolic Method Verifier (SMV) [3], Z/EVES [4], etc.

This paper proposes a novel methodology that verifies the stability of security software. To offer a concrete example, An Access Control System (ACS) is used as a case study. We specify our ACS model in Z and use an automatic translation tool that generates Z schema skeletons corresponding to a UML class diagram created by Rational Rose. The transformation process is not entirely automated since one still needs to annotate a class diagram to fill in some Z-specific details. Once a Z specification is produced, a theorem prover for Z, called Z/EVES is used to formally analyze the semantics of our ACS example and verify its validity. Through this exercise, we demonstrate how one can use our formal specification and verification methodology to develop a vulnerability-free security software system.

The organization of this paper is as follows. Section II discusses related research on formal methods and tools supporting software verification. Section III explains our methodology. Finally, section IV rounds out this paper with some concluding remarks.

2. RELATED RESEARCH

Formal methods are based on mathematics and logic, and used for specifying and verifying both hardware and software systems. One can minimize the ambiguity and uncertainty inherent in natural language specifications by describing systems and their salient features using mathematical notations and logical expressions. Once specified formally, a design can be mathematically verified against user requirements. After this verification process, an implementation of a complex system becomes much more credible.

Formal methods are further categorized into formal specification and formal verification. Formal specification describes requirements for a system, its operational environment, and design using mathematical logic. Requirements and design specifications are two different forms of formal specification. As its name suggests, a requirements specification defines what a system is expected to do while a design specification rigorously explains how a system needs to be built. Formal verification proves completeness or the absence of self-contradictions, and checks whether a design specification satisfies all the requirements by using proof methods available in mathematics and logic.

Manuscript received September 5, 2006.
Manuscript revised September 25 , 2006.

There exist many software verification tools. For example, SPIN is an open-source application that can be used for the formal verification of distributed systems and communication protocols for their correctness and reliability [5], [1]. The on the fly feature of SPIN avoids the use of a pre-built verification plan and improves memory performance. SPIN models the behavior of a given system using automata in a language called PROMELA (PROtocol MEta LAnguage). In PROMELA, a system is represented by parallel processes communicating and synchronizing with one another through established channels. The creators of SPIN developed FeaVer to improve precision and automate source code verification. FeaVer was originally used to check the correctness of telephone call processing code “against a database of formally specified logical correctness requirements” [6].

SLAM is an on-going, software verification tool project run by Microsoft. Targeted mainly for C programs, SLAM does not require any human intervention in extracting models. Instead, it automatically extracts a Boolean program abstracted from a C code implementation and verifies it [2].

SMV verifies a Computational Tree Logic (CTL) specification of software system properties. CTL is a temporal logic based on finite state machines. SMV can specify a system as a synchronous Mealy machine or an asynchronous network. Data types supported by SMV are finite (for instance, Boolean, scalar, fixed arrays, etc) since the tool is designed for finite state machine models. Static and structured data types can also be handled by SMV [3].

Statechart is a graphical language for specifying reactive systems. It introduces, to a state transition diagram, the concepts of hierarchy, concurrency, and communication. A hierarchy in Statechart allows states to have their sub-states, similarly with the relationship between trees and their sub-trees. Concurrency provides states with parallelism. Two or more parallel components can constitute a state. Communica-tion reflects multiple system components exchanging data via broadcasting [7].

Specification and Description Language (SDL) is another language targeted for specifying distributed and reactive sys-tems (as in telecommunications systems). SDL is standardized by International Telecommunications Union (ITU) as Recommendation Z.100. SDL is a multi-purpose language capable of graphical representation/editing, syntactic/semantic error detection, C source code generation, and various types of simulation. SDL can accurately describe functional protocol behaviors by offering structural concepts, design optimization, flexibility of implementation, and techniques to increase mutual understanding between designers. In addition, it features a system analysis ability during a design phase [8].

Security Protocol Engineering and Analysis Resource (SPEAR) is a security protocol development tool. The tool comes with a Graphical User Interface (GUI) that incorporates Message Sequence Charts (MSCs). Once a protocol design is complete, SPEAR can automatically generate Java source code [9].

The Vienna Development Method (VDM) is a set of formal software specification and development techniques. It is composed of (1) a specification language referred to as VDM-SL, (2) refinement rules that create traceability among requirements, design, and source code, and (3) a proof theory by which the correctness of the design and implementation can be verified. The IFAD VDM-SL Toolbox is a VDM-based development environment supporting extensive software verification [10].

Although providingmechanical inspections and user-friendly presentation of proofs, Protocol Verification System (PVS) is not as mature as other well-known verification tools. PVS focuses on creating human readable specifications of domainspecific and mission-critical systems. PVS consists of a specification language and a theorem prover [11], [12].

3. METHODOLOGY FOR SECURITY SOFTWARE VERIFICATION

In this section, we discuss our formal method-based methodology to specify and verify security software. Figure 1 shows the overall process and artifacts involved in using our methodology.

Note that one can take advantage of automated code generation tools such as SPEAR and VDL-SL Toolbox (discussed in section II) to develop security software. If this is the case, a formal design specification for a desired application needs to be fed into the tool before the automatic code generation.

As depicted in figure 1, it is also a probable scenario for one to begin the verification process with a legacy software application. Out of the existing code, a formal design specification can be extracted. Here the use of a formal method is necessary if a reverse engineering tool helps the code extraction effort.

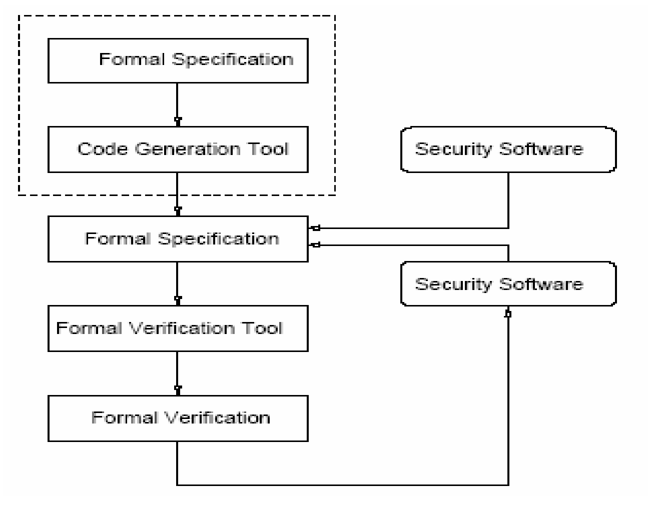

Fig. 1. Security software specification and verification based on the proposed

The security of the code is then verified by a formal verification tool that analyzes the correctness of the derived design specification.

If flaws are found during the verification step, changes are made in the formal specification, and the corresponding, revised code is either automatically (when using a code generation tool) or manually generated, which will be scrutinized again by the verification tool. This cycle continues until there is no more error. As a result, developers can be sure that they have reliable software satisfying all the security requirements at the end of this iterative process.

In the following sub-sections, we demonstrate, by example, how to conduct the iterative verification steps explained so far in the context of security software (more specifically, ACS).We assume the existence of legacy code in our example.

A. Creating a UML model from the existing source code

We first use the reverse engineering feature of Rational Rose C++ (version 4.0) to extract a UML model. Rational Rose C++ has a stand-alone C++ code analyzer program. The automatically generated models represent both logical and physical aspects of the source code. Figure 2 summarizes the step-by-step process for reverse engineering in Rational Rose C++.

B. Transforming a UML model into a Z specification

Once the UML model is available, one can transform it into a Z specification. Z is a formal specification language based on the set theory [13]. Rational Rose can do the conversion with the aid of a RoZ script. RoZ (pronounced as “Rosettee”) can be integrated into the Rational Rose environment, “translate the structure of UML class diagrams into Z specification skeletons, and fill the Z specification with several annotations of the class diagram” [14]. Shown below (figure 3) is a sample UML model
(a portion of an ACS design), which we will use to explain how this transformation occurs.

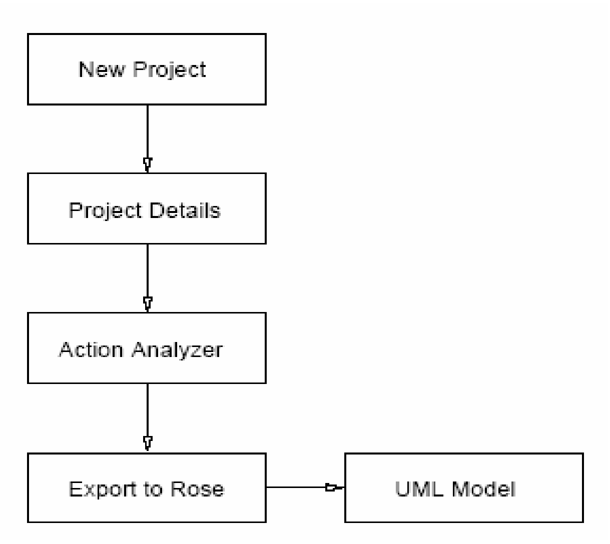

Fig. 2. Extracting a UML model from source code using the C++ analyzer

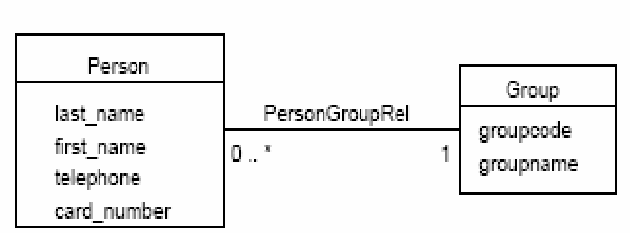

Fig. 3. A partial UML model for an ACS

In the model, each person can be related to only one unique group. Additional restrictions for the relationship (`Person-GroupRel`) can be enforced to improve security,
which include:

1. Every person has at least one telephone number.
2. A card number is a unique identifier for a person.
3. For a given group, everyone in it has the same prefix.

Using the above restrictions, each class in the UML model(figure 3) must be annotated with Z notations before the model can be turned into a Z specification.

For example, the `Person` class needs to be annotated with the following constraint specified in Z.

∀≺1□≺2 : □□→←◦□|≺1 6= ≺2@ (1)
≺1□⇕↦→□ □−∓↕□→ 6= ≺2□⇕↦→□ □−∓↕□→

Equation (1) states that people with two different card num-bers cannot be the same person. That is, two people can never have the same card number. Therefore, this constraint is the Z equivalence to the rule 2: a card number is a unique identifier for a person.

While the constraint in equation (1) is applied to a class itself, the attributes of the class are sometimes subject to one or more Z annotations. For instance, to enforce the rule 1 (every person has at least one telephone number), the `telephone` attribute of the `Person` class needs to be annotated with equation (2) shown below.

```
\begin{zed}
[NAME,TEL]\also
DIGIT8 == 0 \upto 99999999
\end{zed}

\begin{zed}
[GROUPCODE,GROUPNAME]
\end{zed}

\begin{zed}
[PREFIX]
\end{zed}

\begin{axdef}
prefix : TEL \surj PREFIX
\end{axdef}
```

Fig. 4. Type file definitions

↔□ℓ□≺ℏ◦□□ 6= □ (2)

For the third rule, the relevant Z constraint is expressed by equation (4).

∀≺1□≺2 : □□→←◦□| (3)

□→◦−≺˙·.≻□□→←◦□(≺1) = □→◦−≺˙·.≻□□→←◦□(≺2)@

∀↔1 : ≺1□↔□ℓ□≺ℏ◦□□@∀↔2 : ≺2□↔□ℓ□≺ℏ◦□□@

≺→□≻□□(↔1) = ≺→□≻□□(↔2)

Although RoZ can automatically generate a Z specification, the resulting specification is not perfect. This implies that it is often necessary to add additional constraints, types, and properties to the machine-generated version.

For example, data types used in the Z annotations (`NAME`, `TEL`, `DIGIT8`, `GROUPCODE`, and `GROUPNAME`) must be de-fined separately in the form of a Z type file (figure 4).

To generate a correct and complete Z specification, the following conditions are to be satisfied:

- each attribute of a class shall have a type associated with it,
- each operation shall have at least one salient feature, and
- rules for relations shall be specified.

C. Verification of the Z specification

The Z specification of the source code enables one to verify whether the source code fulfills all the security requirements. The Z/EVES [4] tool can be used for this purpose since it can check the consistency of the specification as well as prove theorems mapped to a set of original requirements.

Although most of these theorems require manual creation, RoZ can help automate the generation of some rudimentary theorems related to the base (or basic) operations of a class, which include attribute modifications and the construction/destruct-tion of objects. RoZ can produce the base operations without any human intervention.

In order to utilize this feature of RoZ, the Z type definitions discussed in section B should address guard conditions that can be tied to relevant (security) requirements. For instance, based on one of the guard conditions, RoZ can develop a theorem representing a precondition for an operation called PERSONChangeTel as shown below.

```
\begin{theorem}{PERSONChangeTel\_Pre}
\forall PERSON
; newtel? : \finset TEL
| newtel? \neq \emptyset
@ \pre PERSONChangeTel
```

```
\end{theorem}
```

The theorem states that the input for the operation (namely, `newtel`) belongs to a finite set whose type is `TEL`, and the input is not optional. Z/EVES takes this theorem and proves it against the Z specification prepared earlier.

IV. CONCLUSION

Many tools are available for software verification. Due to their complexity, a majority of these tools involve a steep learning curve. This paper picks Z/EVES as its formal verification tool of choice. The main objective here is verifying whether an implementation conforms to its original requirements. For this, the source code is reverse-engineered into a UML model which is, in turn, transformed into a Z specification. Using a set of theorem proving facilities in Z/EVES, one can compare the extracted Z specification and the formal requirements specification produced when the software was first developed. One of the main contribution of this paper is showing the feasibility of utilizing software verification tools to prove the reliability and dependability of software.

References

[1] G. J. Holzmann, *The SPIN Model Checker-Primer and Reference Manual* Addison-Wesley Professional, Sept. 2003.

[2] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Proc. the 13th Conference on Computer Aided Verification (CAV'01)*, July 18–22, 2001. [Online]. Available: http://research.microsoft.com/slam/

[3] Specification and Verification Center, "The SMV system," 2006.Available:http://www.cs.cmu.edu/_modelcheck/smv.html.

[4] M. Saaltink, "The Z/EVES system," in *Proc. the 10th International Conference of Z Users (ZUM '97)*. Springer-Verlag, Apr. 3–4, 1997, pp. 72–85. [Online]. Available: http://citeseer.ist.psu.edu/saaltink97zeves.html

[5] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997.

[6] Lucent Technologies, "The FeaVer feature verification system," 2006. [Online]. Available: http://cm.bell-labs.com/cm/cs/what/feaver/

[7] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293 – 333, Oct. 1996.

[8] SDL Forum Society, "What is SDL?" 2006. [Online]. Available: http://www.sdl-forum.org/SDL/index.htm

[9] J. P. Bekmann, P. de Goede, and A. C. M. Hutchison, "SPEAR: a security protocol engineering and analysis resource," in *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 3–5, 1997.

[10] J. Fitzgerald, "Information on VDM and VDM++," Jan. 2005. [Online]. Available: http://www.csr.ncl.ac.uk/vdm/

[11] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial introduction to PVS," in *Proc. Workshop on Industrial-Strength Formal Speci-fication Techniques*, Apr. 5–8, 1995.

[12] S. Owre, N. Shankar, and J. M. Rushby, *User Guide for the PVS Specification and Verification System*. CSL, 1995. [Online]. Available: http://citeseer.ist.psu.edu/owre93user.html

[13] A. Diller, *Z: an Introduction to Formal Methods*, 2nd ed. John Wiley & Sons, 1994.

[14] Y. Ledru, "Using Jaza to animate RoZ specifications of UML class diagrams," in *Proc. the 16th International Conference of Z Users (ZUM '06)*. IEEE, Apr. 2006

Seung-Ju, Jang received a B.Sc. degree in Computer Science and Statistics, and M.Sc. degree, and his Ph.D. in Computer Engineering, all from Busan National University, in 1985, 1991, and 1996, respectively. He is a member of IEEE and ACM. He has been an associate Professor in the Department of Computer Engineering at Dongeui University since 1996. He was a member of ETRI(Electronic and Telecommunication Research Institute) in Daejon, Korea, from 1987 to 1996, and developed the National Administration Multiprocessor Minicomputer during those years. His current research interests include fault-tolerant computing systems, distributed systems in the UNIX Operating Systems, multimedia operating systems, security system, and parallel algorithms.

Jungwoo Ryoo is an Assistant Professor in Information Sciences and Technology at Penn State Altoona, Pennsylvania. His main research interests include software engineering, computer networking and telecommunications, and information assurance. More specifically, he is interested in software architecture, Architecture Description Languages (ADL), object-oriented software development, formal methods, requirements engineering, network management, internet security, and e-government. He has a significant industry experience in architecting and implementing software for large-scale network management systems. He received his Ph.D. in Computer Science from the University of Kansas in 2005.

ChangYeol, Lee received a B.Sc. degree and M.Sc. degree from Korea University, and his Ph.D. from University Paris VII in 1985, 1991, and 1997, respectively. He has been an assistant professor in the department of computer engineering at Dongeui University since 2000. He was a member of ETRI(Electronic and Telecommunication Research Institute) in Korea, from 1987 to 1994, and developed AI applications during those years. His current research interests include the digital rights management.