College of Engineering and Computer Science
UNIVERSITY OF CENTRAL FLORIDA

Specifying Security Properties of Protocols in the Java Modeling Language: Achieving Code Level Assurance

Luke Myers luke.myers@knights.ucf.edu
Gary T. Leavens Leavens@ucf.edu
Department of Computer Science

ProVerif

```
out(c, (xA, hostX));
```

ProVerif is an automatic crypotographic protocol verifier [1] which accepts a description of a cryptographic protocol written in the syntax of a typed pi calculus which includes built in cryptographic primatives. ProVerif can securely prove strong secrecy [2] and authentication [3].

JML Specifications

```
requires  sent == 0 && received == 0;
ensures   this.xA.equals
(getIdentifier1Send1(\result)) &&
this.hostX.equals(getIdentifier2Send1
(\result)) && sent == 1;
```

The Java Modeling Language (JML) [5] is a behvaioral interface specification language that specifies the behavior of Java modules with preconditions (requires) and postconditions (ensures) contracts. JML allows the use of "model only" fields to specify additional behavior, e.g. the variables `sent` and `recieved`.

Our Contribution: The Translation and Verification Process

ProVerif to JML Specifications

We start with the formal specification that has been verified by the ProVerif tool. We developed a compiler, using the ANTLR parser generator [9], that parses the typed pi calculus file used by ProVerif. The compiler recognizes sending a message, denoted by the out keyword, and examines the type of the items being sent to determine the types assigned to the generated data fields. Additionally, the compiler creates a method responsible for sending this message and generates JML specifications that ensure the result from this method is equivalent to what is being sent in the formal protocol specification.

JML Specifications to Development

The developer fills in the automatically generated Java class skeleton by implementing the method stubs. The JML specifications serve as the contract the Java code that implements the methods must adhere to but outside of this requirement the developer is free to design the implementation in any way they wish. This allows the developer to optimize the data structures and fine details of the implementation for goals such as better integrating it into a larger application or extracting more performance from it.

Type Checking to Verified Code

The developer uses the OpenJML tool [8] to verify that their implementation satisfies the constraints specified in the JML statements. OpenJML is tool which uses a "Satisfiability Modulo Theory" [6] solver to generate a mathematical proof that the Java code satisfies the constraints described. It is possible for the OpenJML tool to be unable to generate a proof for a particular implementation and the developer will have to return to the Development phase and alter the implementation. After creating an altered implementation, the developer proceeds through the workflow and sees if the OpenJML tool can generate a proof for the new Java code.

Development to Type Checking

The developer annotates their code using standard Java annotation syntax with the annotation of @Hidden to indicate that variable is secret and should not be transmitted unless otherwise specified in the formal protocol specification or it has been encrypted. The developer uses the annotation of @Open for all other variables to indicate that they are not critical to the security of the system and can be transmitted at any time. The type annotations for the data fields automatically generated by our compiler from the formal protocol description are also automatically generated by our compiler. The types are checked with a custom module in The Checker Framework [7].

Verified Code

```
requires  sent == 0 && received == 0;
ensures   this.xA.equals
(getIdentifier1Send1(\result)) &&
this.hostX.equals(getIdentifier2Send1
(\result)) && sent == 1;

@Open BitSet send1(@Open Identifier xA,
@Open Identifier hostX) {
/* ...implementation... */

}
```

Once the OpenJML tool is able to verify the code, the process is complete. The Java source code file, with the type annotations and JML specifications, is compiled by the standard Java compiler into executable byte code.

Type Checking

```
requires  sent == 0 && received == 0;
ensures   this.xA.equals
(getIdentifier1Send1(\result)) &&
this.hostX.equals(getIdentifier2Send1
(\result)) && sent == 1;

@Open BitSet send1(@Open Identifier xA,
@Open Identifier hostX) {
/* ...implementation... */

}
```

These security type annotations describe which information should be kept secret (`@Hidden`) and which should be allowed to be transmitted (`@Open`). Our custom type checker analyzes these types to check for accidental leakage of secret information.

Overview

Background: It is vital that computer communication protocols are not only securely designed but also securely programmed. The importance of this problem is exemplified by a recent bug called Heartbleed found in the Transport Layer Security protocol, which is responsible for securing communication between websites and web browsers, that made vulnerable an estimated 24-55% of the 1 million most popular websites [4].
Problem: Verify that the implementation (code) of a cryptographic protocol is performing the protocol in accordance with the security properties of the formal specification of that protocol.
Approach: We build upon an existing tool which verifies formal protocol specifications and compile the specification into implementation language specifications which are then verified by our type checker and an existing implementation language verifier.
Contribution: Existing approaches to this problem fall into two categories. The first category is comprised of solutions that take a protocol specification and translate directly from that specification into a fully implemented program. The second category is comprised of solutions that take a fully implemented program and translate it into a protocol specification which they then attempt to prove is equivalent to the reference protocol specification. Our approach falls into a novel third category of compiling a protocol specification into a set of implementation language specifications. Compared to existing solutions that generate the completed program from the protocol specification, our tool allows more flexibility in the exact implementation details of the program while avoiding the unreliability of attempting to translate a program into a protocol specification that is equivalent to the reference.

Development

```
requires  sent == 0 && received == 0;
ensures   this.xA.equals
(getIdentifier1Send1(\result)) &&
this.hostX.equals(getIdentifier2Send1
(\result)) && sent == 1;

BitSet send1(Identifier xA, Identifier
hostX) { /* ...implementation... */


}
```

The developer writes code to implement the methods in the Java class skeleton generated by our compiler. This is contrary to existing solutions as the developer writes the code instead of the compiler, so the developer is free to optimize the code for efficiency or other concerns without voiding the security guarantee.

References

[1] Blanchet and Bruno. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules, 2001.
[2] B. Blanchet. Automatic proof of strong secrecy for security protocols. *In IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 86–100. IEEE.
[3] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. pages 342–359. Springer, Berlin, Heidelberg, 2002.
[4] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter of Heartbleed. IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.
[5] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. *ACM SIGSOFT Software Engineering Notes*, 31(3):1, may 2006.
[6] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979
[7] MM Papi, M Ali, TL Correa Jr, and JH Perkins. Practical pluggable types for Java. *Proceedings of the 2008*, 2008.
[8] OpenJML. http://www.openjml.org/
[9] TJ Parr and RW Quong. ANTLR: A Predicated. *Software—Practice and Experience*, 1995.