2012

Deductive formal verification of embedded systems

Zheng Lu
Louisiana State University and Agricultural and Mechanical College, zlu5@tigers.lsu.edu

DEDUCTIVE FORMAL VERIFICATION OF EMBEDDED SYSTEMS

A Dissertation

Submitted to the Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Zheng Lu
B.S., in Computer Science, Central South University, PR.China 2002

December 2012

Acknowledgments

This dissertation would not be possible without several contributions. It is a pleasure to thank many people who made this dissertation possible. I will remember and appreciate their contributions forever.

First and foremost, I would like to deeply thank Prof. Supratik Mukhopadhyay. He has always encouraged me to overcome the many difficulties faced during my research, and has supported me with his professional knowledge. Only with his support could I complete this dissertation.

I would also like to thank the other members of my dissertation committee, Prof. Jianhua Chen, Prof. Konstantin Busch ,Prof. Gary R. Byerly and Prof. Phuc Nguyen. They have provided valuable feedback and interesting perspectives on the ideas contained in this dissertation. Once again I would like to thank them all for being on my committee.

Finally, I would give great appreciation to my family. They always gave me unquestioning faith and encouraged me in every time. Especially, I would love to express to Wen Jiang, my wife, my gratitude for her enormous support and devotion. Without her sacrifices, I could not have completed this long journey. So, I would like to dedicate this dissertation to her.

Contents

List of Tables

List of Figures

Abstract

We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework that, given an analysis aspect of the source code, automatically generates an analyzer capable of inferring information about that aspect.

The analyzer is generated by translating the collecting semantics of a program to a formula in first order logic over multiple underlying theories. We import the semantics of the API invocations as first order logic assertions. These assertions constitute the models used by the analyzer. Logical specification of the desired program behavior is incorporated as a first order logic formula. An SMT-LIB solver treats the combined formula as a constraint and solves it. The solved form can be used to identify logical and security errors in embedded programs. We have used this framework to analyze Android applications and MATLAB code.

We also report the formal verification of the conformance of the open source Netgear WNR3500L wireless router firmware implementation to the RFC 2131. Formal verification of a software system is essential for its deployment in mission-critical environments. The specifications for the development of routers are provided by RFCs that are only described informally in English. It is prudential to ensure that a router firmware conforms to its corresponding RFC before it can be deployed for managing networks. The formal verification demonstrates the usefulness of inductive types and higher-order logic in software certification.

Chapter 1

Introduction

Software programs can be uncertified and may contain malicious code or vulnerabilities that can be exploited to leak confidential data, perform pernicious activities, and destroy or modify valuable information. Hence, the correctness and reliability of these systems software have become issues of utmost importance. In this chapter, we first give a brief introduction on program analysis and formal verification techniques. Then, we describe our motivation and objectives of this dissertation.

1.1 Program Analysis

Software systems routinely manage mission-critical activities in organizations that rely on dependable, situation-aware, and timely delivery of classified or sensitive information. Information flows in such enterprises are processed by custom-built, open-source, software programs. These software programs can be uncertified and may contain malicious code or vulnerabilities that can be exploited by an insider or an outsider to leak confidential data, misclassify documents, perform pernicious activities, and destroy or modify valuable information. Hence, the correctness and reliability of software driving these systems have become issues of utmost importance.

Application-independent errors in software systems like buffer overflows and null dereferences can be exploited by malicious applications to create security holes through which confidential data can be leaked. Many of these bugs are not detected until much later when catastrophic effects are already visible [83] making difficult the task of runtime fault handling mechanisms for ensuring recovery. A more important concern is the lack of proper tool support for detecting logical application-dependent errors in programs. An examination of a list of well known incidents resulting from software glitches reveals that application-dependent logical errors were the causes of most [20] [11] [8] (e.g., the USS Yorktown breakdown and the failure of the Patriot missile). Many of these logical errors were deep (as opposed to simple typos) and are difficult to detect using state-of-the-art testing techniques alone [32].

While model checking techniques [39] have been able to uncover deep logical errors in hardware [34], the success has not been carried forward to the domain of software. One reason for the lack of success of software model checking is the infinite state nature of software. Most well-known program analysis tools (e.g., Uno [58], Splint and LCLint [48], Polyspace [16], Codesurfer [26], PREfix and PREfast [49], and ESP [44]) perform lightweight data flow analysis. Tools like CCured [72] analyze programs to determine a set of dynamic checks that can prevent memory leaks. Logical errors such as those reported in [20] [11] [8] evade such analyzers. On the other end of the spectrum, tools such as CMC [71], MAGIC [34], CBMC [40], and SLAM [28] perform finite state model checking of programs. These tools first generate a finite model from the source code using predicate abstraction techniques (while SLAM, CBMC, and MAGIC generate the model a priori, CMC generates the model on the fly). Analysis is then carried out on this finite model (similar to hardware model checking). Once verified, this finite

model is discarded; the effort spent developing it is wasted. Furthermore, it is difficult to relate the analysis (counterexamples) obtained at the abstraction level of the finite state model to the source code. Finite state analyses (as in CBMC, SLAM and MAGIC) tend to lose information before the analysis even starts. They do not take into account that in many cases infinite state analysis might terminate in a reasonable amount of time without losing any information.

Program analysis tools like Coverity [5] perform path simulation and interprocedural dataflow analysis through every method in the code for C, C++, Java, and C♯. However, typically the source code is unavailable for a large number of methods; hence such techniques will not be able to deal with deep security flaws resulting from subtle interactions of these methods with the rest of the code. Tools like JavaPathFinder [75] simulate Java code up to a certain depth and hence are likely to miss security flaws that are usually deep.

1.2 Formal Verification and Certification

Formal certification of software has been the holy grail of formal methods and verification research for several decades [21]. While this has been the original aim of the field of formal methods, and a tremendous amount of effort has been invested in creating practical tools for proving the correctness of software systems since the 1970's, in the initial years, success was limited to deductive verification of toy examples with a few lines of code [68]. Despite the fact that formal verification techniques like theorem proving are routinely being used in hardware, their failure to scale in case of software has prevented the evolution of a formal codesign framework. The lack of practicality has led to the effort getting diverted to developing techniques and tools that can detect errors in models of software systems [39] rather than formally verifying them. Recently

however, there has been a tremendous amount of interest in formally certifying software that has been spurred by several unprecedented successes in this area [67, 63, 65]. These successes have been achieved by leveraging the accumulated knowledge and the tools that have been developed in the formal methods community over the past three decades and applying them to ensure high assurance in significantly large software systems.

1.3 Motivation and Objective

Google Android provides an open source customizable platform for Google Android mobile phones. Android applications are written in the Java programming language. Since applications that run on a phone can be written by any developer, there is always a possibility that programming flaws introduced inadvertently or maliciously can be exploited by malicious individuals. For example, a malicious individual can inject code that will enable an application downloaded on a phone to make calls to an international number exploding the phone bill to thousands of dollars.

While Androids security model can prevent an application, e.g., from making phone calls, once an application has been given that permission, the security model is not fine-grained enough to prevent all malicious activity. Hackers also may be able to use malicious code to remotely control the phone and use the private data stored in it for malicious purposes. A recent vulnerability related to Android involves using a malformed SMS to cause a Java ArrayIndexOutOfBounds exception resulting in the phone getting disconnected from the network. Uncertified programs running on the Android platform may contain vulnerabilities that can be exploited to leak confidential data, misclassify documents, and destroy or modify valuable information.

We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework that, given an analysis aspect of the source code, automatically generates an analyzer capable of inferring information about that aspect. A model-based technique is necessary since for many of the APIs invoked as well as the objects instantiated, the (source) code is not available. The only information that the analyzer can get about the properties of these artifacts is from their models. The analyzer is generated by translating the collecting semantics of a program to a "marked" formula in first order logic over multiple underlying theories. The "marking" can be thought of as a set of holes or contexts corresponding to the "uninterpreted" APIs invoked in the program. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions. These assertions constitute the models used by the analyzer. Logical specification of the desired program behavior (rather its negation) is incorporated as a first order logic formula. An SMT-LIB formula solver treats the combined formula as a "constraint" and "solves" it. The "solved form" can be used to identify logical (security) errors in Java (Android) programs.

The Netgear WNR3500L router is the first product, encouraged in industry, for which customers can choose between the manufacturer's router firmware and several open source alternatives. Netgear has come to recognize that there are many customers who transform low-cost router equipment into high-end network devices by using advanced firmware. They openly support and encourage people to publish their updated firmware [12]. It is essential to ensure that a router firmware contributed by the open source community conforms to its corresponding RFC before it can be deployed for managing mission-critical networks.

In the Netgear forums, users complained that the Netgear WNR3500L wireless router has problems such as frequently getting IP address allocation errors, etc., it is clear from reading the RFC that such malfunctions should not have occurred if the RFC was strictly adhered to. This served as our principal motivation in attempting to formally verify the conformance of the implementation with respect to the RFC.

We report the formal verification of the conformance of the open source C implementation of the Netgear WNR3500L wireless router firmware to the RFC 2131 [24] based on which it is designed. The C source code of the router firmware consists of 2580 lines of C code. The formal verification effort led to the discovery of several possible problems in the implementation. We have used the Coq proof assistant extensively in this verification effort.

Chapter 2

Related Works

Techniques for software verification and validation fall into three main categories. The first category involves informal methods such as software testing and monitoring. Such techniques scale well; this is by far the most used technique in practice to validate software systems. Testing accounts for forty to sixty percent of the development effort [32] [82]. Traditional software testing methods [41], however, are too ad hoc and do not allow for formal specification and verification of high-level logical properties that a system needs to satisfy. In the realm of safety critical software where exponential blow up in the number of possible situations to be dealt with is inevitable, traditional testing techniques can hardly be used to provide any amount of confidence. The second category of techniques for software verification and validation involves formal methods. Traditional formal methods such as model checking [39] and theorem proving [35] are usually too heavy and rarely can be used in practice without considerable manual effort.

Model checking is an automatic approach to verification, mainly successful when dealing with finite state systems. It not only suffers from the infamous state explosion problem but also requires construction of a model of the software. Such a construction effort not only requires skill and ingenuity in model building but also a deep understanding of the operational underpinnings of the target software. Theorem proving is not only labor intensive but also requires considerable skill in formal logic.

The third category of techniques for software verification and validation are static analysis [73] and abstract interpretation [42]. Static analysis refers to the technique(s) for automatically inferring a program's behavior at compile time. While static analysis tools have met with tremendous practical success and have been routinely integrated with state of the art compilers, such tools can only detect shallow and simple errors due to their lack of deductive power. For example, traditional static analysis tools cannot detect the presence of deadlocks or the violation of mutual exclusion in concurrent programs. Abstract interpretation is a technique for collecting, comparing, and combining the semantics of programs. It has been successfully used to infer run time properties of a program that can be used for program optimization. The next few paragraphs review the most successful approaches to program analysis.

2.1 Static Code Analysis

In recent years, much work has been done on static analysis of software. Some static analysis tools, such as Uno [58], Splint [48], Polyspace [16], Codesurfer [26], PREfix and PREfast [49], ESP [44], and PAG [69] perform lightweight data flow analysis. Coverity [56] performs data flow analysis as directed by checkers written in MetaL, a language designed to encode checking automata. Astree is a static program analyzer that is aimed at proving absence of runtime errors in embedded programs. Astree can handle only a "safe" subset of C, rather than the full C language. Also, it applies only to particular runtime errors rather than general properties of programs. Halbwachs et al [55] use linear relation analysis for discovering invariant linear inequalities among the numerical variables of a program. Their techniques have been used to validate (e.g., analyze delays) in synchronous programs written in the language Lustre. Several

abstractions have been considered to provide an approximate (conservative) answer to the validation problem such as widenings, convex approximations and Cartesian factoring [54]. These approximations are implemented using the polka [55] polyhedral library. Alur et al [25] have used predicate abstraction for analyzing hybrid systems. In this technique, a finite abstraction of a hybrid automaton is created a priori using the initial predicates provided by the user. Set based techniques for detecting races in relay ladder programmable logic controllers have been described in [23]. Context-sensitive analysis using deductive database techniques [64] are similar to ours. However, this technique alone is insufficient to achieve the goals we aim for due to the limited expressiveness of Datalog. Typed assembly languages help detect security flaws in code. However, it is difficult to provide any insight to the developer in the event of such detection.

Tools like SofCheck Inspector [19] inspect every method of Java programs and compute their pre and post conditions. However, for many packages source code is not available. Findbugs [27] analyzes Java byte code and detects bugs due to common programming mistakes based on bug patterns. However, it is difficult to provide any meaningful insight to the developer from bugs found at the byte code level. Besides, it is difficult to provide bug patterns for deep logical errors. The program analysis tool that uses an approach closest to ours is Fortify's [7] source code analysis engine based on verification condition generation. However, unlike our approach, their tool cannot automatically generate a specialized analyzer for a particular aspect. Boon [79] uses range analysis techniques to check for array bounds violations in C programs. However, it is not able to deal with "high level" languages like Java where typically a program is a chain of method invocations. Chaudhuri [36] describes a language-based approach for ensuring security in the Android platform. However, his type-based analyzer provides a coarse-grained analysis, is

not directed towards Java application code. Fuchs, Chaudhuri, and Foster [52] provide a program analysis framework for Android applications. They use a core calculus for modeling Android applications. However, it is difficult to incorporate deductive techniques into their framework for deep logical analysis. Klocwork provides a static analysis framework for Java intended for the Android platform [10]. However, unlike the presented framework, their framework is not model based. Kirin [47] provides a policy-driven lightweight security certification service that can certify Android apps before they are downloaded by examining the manifest. In contrast our framework statically analyzes source code and tries to infer "deep" vulnerabilities. Jif [9] is a tool for guaranteeing noninterference properties in Java programs. In contrast our framework uses model-based deductive static verification to uncover bugs in Android apps.

Techniques for system verification and validation fall into three main categories. The first category involves informal methods such as testing and monitoring [32] [82]. Such techniques scale well; they are extensively used in practice to validate systems. Traditional testing methods [41], however, are too ad hoc and do not allow for formal specification and verification of high-level logical properties that a system needs to satisfy. In the realm of mission-critical systems where exponential blow up in the number of possible situations to be dealt with is inevitable, traditional testing techniques can hardly be used to provide any amount of confidence. The second category of techniques for verification and validation involves formal methods. Traditional formal methods such as model checking [39] and theorem proving [35] are usually too heavyweight and rarely can be used in practice without considerable manual effort.

Model checking is an automatic approach to verification, mainly successful when dealing with finite state systems. It not only suffers from the infamous state explosion problem but also

requires construction of a model of the system. Such a construction effort not only requires skill and ingenuity in model building but also a deep understanding of the operational semantics of the target system. Theorem proving approaches are not only labor intensive but also requires considerable skill in formal logic.

The third category of techniques for software verification and validation are static analysis [73] and abstract interpretation [42]. Static analysis refers to the technique(s) for automatically inferring a program's behavior at compile time. While static analysis tools have met with tremendous practical success and have been routinely integrated with state of the art compilers, such tools can only detect shallow and simple errors due to their lack of deductive power. For example, traditional static analysis tools cannot detect the presence of deadlocks or the violation of mutual exclusion in concurrent programs. Abstract interpretation is a technique for collecting, analyzing, and comparing the semantics of programs. It has been successful in analyzing properties of complex programs [42]. The next few paragraphs review the most successful approaches to program analysis.

2.2 Proof Assistants and Theorem Provers

Many automated proof tools exist based on different forms of higher-order logic such as HOL [84], Coq [4], PVS [17], Isabelle [74], ACL2 [62], and Mizar [53]. Different engines implement different proof systems and provide different input languages for specifying theories and tactics.

Coq is a proof assistant for the Calculus of Inductive Constructions, a higher order logical framework that includes dependent types and a primitive notation for inductive types [50]. It allows interactive construction of formal proofs from higher order theories as well as creation

of provably correct functional programs consistent with their specifications [4]. Coq has been written in the OCaml programming language.

HOL implements a classical higher order logic based on Church's simple theory of types [38] extended with polymorphic types and inference rules for definition. In HOL, specifications are typed in the sense that terms have types, where types can be constants, function types, compound types, or type variables for polymorphism [84].

The PVS proof checker provides a collection of primitive inference procedures including propositional and quantifier rules, induction, rewriting, data and predicate abstraction, and symbolic model checking. Users can combine these primitive inferences with user-defined procedures to yield higher-level proof strategies. PVS includes a BDD-based decision procedure for the relational mu-calculus and thereby provides an experimental integration between theorem proving and CTL model checking [17].

Like HOL, Isabelle allows writing specifications in the constructive portion of Church's simple theory of types. It implements a higher-order flavor of resolution. Automated proof generation techniques like tableaux are also built into Isabelle.

The Mizar proof checker was aimed at formalizing mathematics and for machine-assisted checking of mathematical proofs. ACL2 is an automated theorem prover that implements quantifier-free first order logic. The Nuprl system [31] provides a logical framework based on constructive type theory.

Given such a plethora of formal verification tools, the choice of an appropriate proof assistant for the given verification problem is a difficult task. We believe that using an interactive theorem prover for verifying a software system provides the prover/developer with significant insights into

the system that can be used to uncover defects as well as conceptualize new ways to implement the system. It is difficult without these insights to analyze the system behavior for bugs if the proof fails. Besides the insight gained can facilitate the creation of proper abstractions/refinements that can greatly simplify the verification process.

Most of the proof tools mentioned above have been used for significant industrial strength case studies. We have already stated why we would prefer an interactive proof tool rather than a completely automated one like ACL2 for verifying a system such as router firmware. While other interactive proof tools like HOL, Isabelle, and PVS are good candidates, Coq provides an OCaml-based specification language that enables writing modular specifications. Besides it is easy to reuse proof components in Coq. In addition, like [37], we prefer Coq in our verification effort for the mature proof environment that it provides.

Model checking techniques [39] have been used to uncover bugs in significantly large software designs [29]. However the goal of our effort is not only to uncover non-conformance errors but also to prove the formal correctness of the implementation with respect to the appropriate RFC. Static analysis techniques [73] have been used to uncover application-independent errors (like array bounds violation, buffer overflow, etc.) in C programs. In this verification effort we are concerned with application-dependent errors rather than application-independent ones.

2.3 Software Certification

In [63], seL4, a member of the L4 microkernel, is the first OS kernel that is fully formally verified for functional correctness. The kernel comprises 8,700 lines of C code and 600 lines of assembler. The authors report the construction of a Haskell prototype, which is proved by a refinement proof

in Isabelle/HOL, to model the implementation of the kernel design, then manually re-implement the model in the C programming language.

A compiler needs to determine an appropriate memory layout to store a data structure. Since the semantics of the C++ language gives a flexibility for this memory layout, many optimized object layout algorithms have been proposed. Ramananandro, Dos Reis, and Leroy [77] provide a specific C++ object layout algorithm and prove its correctness against the operational semantics of C++ multiple inheritance as formalized by [81].

Malecha et. al. [67] describe the implementation of a lightweight, fully verified relational database management system. They construct a complete specification of the relational algebra model for defining schemas, relations, and query operation. The SQL abstract syntax and denotational specification are provided. They also implement a B+ tree for insertion and iteration with a run-time cost model and proved that certain transformations do not increase the runtime cost.

Leroy reports the development of a formally certified compiler for a C-like imperative language with the PowerPC assembly code as the target [65]. The formal verification of this compiler proves that safety properties satisfied by the program source code also hold for the compiled code.

In [45], Deng et.al. proposed a general formalization of self-stabilizing population protocols. They proved that the leader-election in this protocol is self-stabilizing for a network of arbitrarily large size.

In [70], Moller translated an informal specification of an asynchronous message router into the formal language of the modal $\mu-$calculus. They then described the implementation of a router in the Calculus of Communicating Systems (CCS). They used the Concurrency Workbench

to verify that the CCS term corresponding to the router implementation satisfied the $\mu-$calculus specification. The commercial router as well as its specification that we consider here are an order of magnitude more complex than the one considered in [70]. Hence a powerful proof assistant such as Coq is necessary for performing such a verification task.

In [61], the authors describe a Hoare-like logic for specifying and verifying protocol layers in communication networks. The possibility of using theorem provers for Hoare-logic to formally verify existing implementations of protocols or build provably correct implementations of protocols is mentioned in the conclusion of this work, though not attempted in the paper itself. In [66], the authors describe the development of a provably correct group communication system called Ensemble using the Nuprl reasoning system. Zave [85] describes a formal framework for describing patterns of identifier binding during communication. In [80], the authors describe a declarative framework for rapid prototyping of network protocols. A theorem prover is used to formally verify declarative specifications of network protocols. In contrast with the above works, we report the formal verification of the conformance of the industrial strength open source Netgear WNR3500L wireless router firmware to the RFC 2131 [24] based on which it is designed.

Chapter 3

Model-Based Static Source Code Analysis of Java Programs

We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework that, given an analysis aspect of the source code, automatically generates an analyzer capable of inferring information about that aspect. The analyzer is generated by translating the collecting semantics of a program to a "marked" formula in first order logic over multiple underlying theories. The "marking" can be thought of as a set of holes or contexts corresponding to the "uninterpreted" APIs invoked in the program. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions. These assertions constitute the models used by the analyzer. Logical specification of the desired program behavior (rather its negation) is incorporated as a first order logic formula. An SMT-LIB formula solver treats the combined formula as a "constraint" and "solves" it. The "solved form" can be used to identify logical (security) errors in Java (Android) programs. Security properties of Android are represented as constraints and the analysis aims to show that these constraints are respected.

3.1 Overview of the Android Platform

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. A central feature of the Android system is that an application can make use of elements of other applications. To accomplish this, the system needs to start a process when any part of an application is needed. Android doesn't have a single entry point (no main() function). They have essential components that the system can instantiate and run as needed. The four main types of components are

1. *Activity:* an application component that provides a user interface with which users can interact with the device, such as dial the phone, take a photo or view a map. Each Android application needs to contain at least one activity.

2. *Service:* an application component that can perform long-running computations in the background and does not provide any user interface. For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider from the background.

3. *Broadcast receivers:* a component that responds to system-wide broadcast announcements. For example, messages may be send to all applications that the system language setting has been changed

4. *Content providers:* application components which are used to store and retrieve data and make it accessible to all applications.

Android uses intent to activate the first 3 components. For activities and services, the intent is the pair: <action name, data>, which indicates the predefined action that the receiver needs to take and the *data* to process. In our program analysis framework, this intent object is our *keyword*, we use this intent object to perform data flow analysis and function call analysis.

3.1.1 Security Mechanism

The Android architecture supports building applications with phone features and protecting users by minimizing the consequences of bugs and malicious software. In Android, an application can share its data and functionality with other applications. These accesses must be controlled carefully for security. Here are some key access control mechanisms in Android.

- Isolation: The two isolation postulates used in the Android platform are
    1. Each Android application runs in its own Linux process.
    2. Each application has its own virtual machine, runs in isolation from the code of all other application.
- Permissions: Any application needs explicit permissions to access other applications. These permissions are set at install time.
- Signatures: An Android application must be signed with a certificate whose private key is held by the developer.

The process isolation alleviates the need for complicated policy configuration files and gives applications the flexibility to use native code without compromising security or granting the application additional rights.

Android permissions are rights given to applications to allow them to perform functions like take pictures, use the GPS, or make phone calls. When applications are installed, they are given a unique UID, and each application always runs under that UID on that particular device. The UID of an application is used to protect its data sharing with other applications.

Android requires users to validate the permission list of programs that can do dangerous things like:

- directly dial calls;
- disclose user's private data, such as access some local files;
- destroy address books, email, etc.

Consider the following code that tries to call a phone number input by a user

```
public class Test extends Activity{
  public void onCreate
    (Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    final EditText phoneNumber
        = (EditText) findViewById
            (R.id.phoneNumber);
    final Intent CallIntent = new Intent
        (Intent.ACTION_CALL,
            Uri.parse("tel:"
            + phoneNumber.getText()));
    callButton.setOnClickListener
            (new OnClickListener(){
      public void onClick(View v){
        startActivity(CallIntent);
      }
    });}}
```

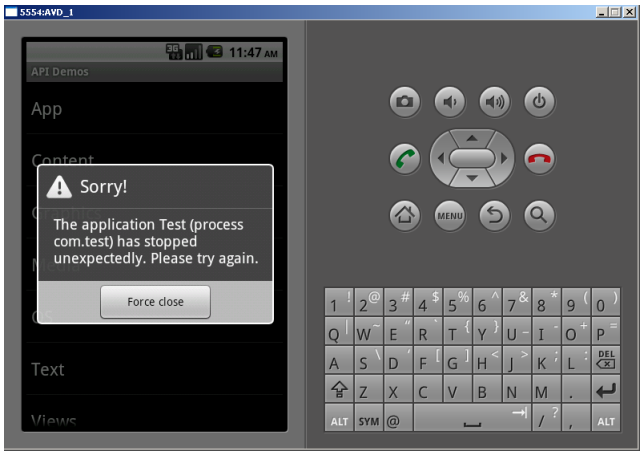

Figure 3.1: Android Runtime Error

Launching of this application results in the error shown in Figure 3.1.

The program constructs an `Intent` named `CallIntent`. Then it uses this intent to start a new activity `Intent.ACTION_CALL` which will call a phone number input from the text box `phoneNumber`. However this application will fail and crash the system because it has no permission to perform this `Intent.ACTION_CALL` action.

Android uses manifest permissions to track what the user allows applications to do. An application's permissions are expressed in its `Manifest.xml` and the user agrees to them upon installation.

Permissions must be associated with some goal that the user understands. For example, an application needs the `READ_CONTACTS` permission to read the users address book. A contact manager application needs the `READ_CONTACTS` permission. If an application requests too few permissions; it is *underprivileged* and easy to detect. If an application requesting more permissions than it needs; it is *overprivileged*. In overprivileged applications an attacker can exploit the unnecessary permissions and takes control of the application.

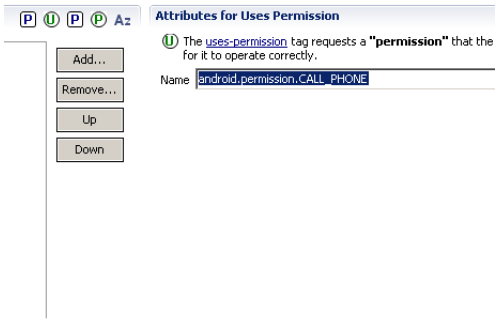

Figure 3.2: Setting Up Permission in Android SDK

3.1.2 Dalvik VM

Android uses Dalvik as the virtual machine to run the Java platform on mobile devices. Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk (Android Package) file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language.

Dalvik is optimized for low memory requirements, and is designed to allow multiple VM instances to run at once, relying on the underlying operating system for process isolation, memory management and threading support. Dalvik is not the standard Java Virtual Machine. The architecture is register-based. A tool called dx is used to convert Java .class files into the .dex format. Multiple classes are included in a single .dex file. Duplicate strings and other constants used in multiple class files are included only once in the .dex output to conserve space. Java bytecode is also converted into an alternate instruction set used by the Dalvik VM. An uncompressed .dex file is typically a few percent smaller in size than a compressed .jar derived from the same .class files.

3.1.3 Architecture of Our Model

Figure 4.1 shows the architecture of our model-based static analysis approach. The abstract collecting semantics of Java programs are represented as "marked" constraints. The "marking"s can be thought of as a set of holes or contexts corresponding to uninterpreted APIs, i.e., library APIs whose semantics are not known. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions or constraints. These assertions are the models that are used to "unmark" the abstract collecting semantics constraints, i.e., "filling in" the "holes" left by uninterpreted APIs. Analysis aspects are specified as constraints. Basic constraint solving is done using a combination of decision procedures provided by the Yices [46] constraint solver.

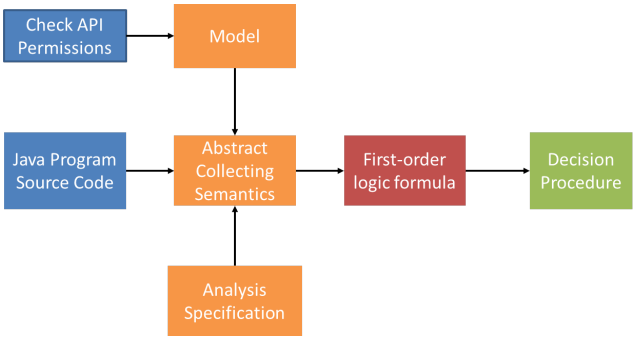

Figure 3.3: Architecture of Our Verification Framework for Java Programs

The key steps involved in our analysis framework are:

1. Verify the permissions of the Android APIs invoked in the Java source code based on the `Manifest.xml`. The results of the permission verification are used to modify the models of the APIs

2. Generate abstract collecting semantics constraints from the Java source code,

3. Import models of uninterpreted methods and objects as assertions into the already generated constraints; uninterpreted methods/objects need to be annotated by the programmer; annotation is needed since a particular method might be overridden by the developer and hence importing its "conventional" model from a model library may result in unsoundness of the analysis, and

4. Generate an analyzer by adding appropriate analysis "aspect" constraints,

5. Analyze by solving the constraints.

To carry out the four steps above, we describe the following technologies and tools that we have developed following tools and use Stowaway [22] as our permission verification tool.

1. An engine for extracting constraints (collecting semantics) from Java programs,

2. An engine for importing model assertions into the extracted constraints,

3. A transformation scheme for weaving in the analysis aspects into the body of the extracted constraints

4. An engine combining decision procedures for solving constraints.

3.1.4 Permission Verification

In this section, we discuss how we verify whether an application has the correct permission settings. In Table 3.1, Android permissions are categorized into different protection levels [33] [78].

Table 3.1: Android Permission Protection Levels

Normal	Permissions for applications whose consequences are minor such as VIBRATE which lets applications vibrate the device.
Dangerous	Permissions, such as WRITE_SETTINGS or SEND_SMS are dangerous since they could be used to reconfigure the device or incur tolls. Android needs to warn users about the need for these permissions on installation.
Signature and System	These permissions can only be accessed by other applications signed with the same keys as this program. This protection is to help integrate system builds and not provided by developers.

Permissions are defined in the `Mainifest.xml` file. Our framework will examine this file and retrieve the permission information. We build a required permission list by analyzing the APIs invoked in a program and compare it with the permission information $\mathcal{P}$ retrieved from the `Mainifest.xml` file. Consider the following example

```
public class CallIntends extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
  public void callIntent(View view) {
    Intent intent = null;
    intent = new Intent(Intent.ACTION_CALL,
        Uri.parse("tel:(+49)12345789"));
    startActivity(intent);
    }
  public void onActivityResult(int requestCode,
        int resultCode, Intent data) {
    if (resultCode == Activity.RESULT_OK && requestCode == 0) {
      String result = data.toURI();
      Toast.makeText(this, result, Toast.LENGTH_LONG);
    }
  }
}
```

This application tries to call a phone number `(+49)12345789`, which is hard coded in the program. To process this action, the application needs a `CALL_PHONE` permission, for example:

```
<uses-permission android:name="android.permission.CALL_PRIVILEGED"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
```

We map the Android API calls to the required permission list using Stowaway [22]. If every required permission is provided we consider that the application has no permission violation. Otherwise, any API calls which have no appropriate permissions provided will be asserted as returning -1 in the analysis that follows.

In this simple example, we cannot find the `CALL_PHONE` permission from `Mainifest.xml`. So there is a permission underprivilege error.

3.1.5 Constraints, SMT-LIB Solvers, and Satisfiability

Constraints are special formulas of first order logic [51]. A constraint system formally specifies the syntax and semantics of constraints. The following definitions are based on [57] and [60].

Definition 1 (Constraint System)**.** *A constraint system is a tuple* (Σ, D, CT, C)*, where*

- Σ*, the signature, is a finite set of function and relation symbols including the constants* $true$ *and* $false$ *and the binary relation symbol* $=$ *for equality.*
- D*, the domain, is a set together with an interpretation of the function and relation symbols in* Σ*.*
- CT*, the constraint theory, is a non-empty set of first order logic formulas (axioms) over the signature* Σ*.*

- *C, specifying the syntax of the constraints, is the set of all (allowed) first order logic formulas closed under existential quantification over the signature Σ that contains the constraints true, false, and =.*

CT defines the semantics and C the syntax of the constraint system.

A constraint solver implements an algorithm for checking *satisfiability/consistency* of a set of constraints using the constraint theory, i.e., determining if there exists an assignment of the variables that satisfies the constraints. A solver uses axioms of the constraint theory together with simplification rules as rewrite rules to transform the constraints to a "normal" form called the "solved form". The final constraint that results from such a computation is called the answer.

Definition 2 (Satisfiability Condition for Constraint Solver)**.** *For a set of constraints $\mathcal{C}$ a constraint solver returns false if $\mathcal{C}$ is inconsistent.*

For example, $X > Y \bigwedge Y > X$ is inconsistent, and $X \geqslant Y \bigwedge Y \geqslant X$ is consistent. A solver implements a *decision procedure* for checking satisfiability of constraints.

3.2 SMT-LIB and Yices

Satisfiability Modulo Theories (SMT) libraries [30] provide a framework for checking the satisfiability of first-order formulas with some background logical theories. SMT-LIB is an SMT library that provides a standard description of the background theories used in SMT systems; it gives a common input and output languages for SMT formula solvers.

An SMT-LIB formula instance is a formula in first-order logic in SMT-LIB syntax, where some function and predicate symbols have additional interpretations, and SMT formula satisfia-

bility is the problem of determining whether such a formula is satisfiable. We can consider SMT satisfiability as an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of variables that range over different domains. The predicates include linear inequalities, such as $3x + 2y - z \geqslant 0$ or equalities involving uninterpreted functions; for example, $f(f(u, v), v) = f(u, v)$ where f is some unspecified function of two unspecified arguments.

The predicates are classified according to the theory they belong to. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, some predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality. Other theories include the theories of arrays and list structures, and the theory of bit vectors.

Yices [46] is an efficient SMT-LIB formula solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions. An example of constraints in the SMT-LIB formula syntax is given below. In this example, we use the theory `QF_LIA`, quantifier-free linear integer arithmetic, to declare two functions which return integer values. The satisfiability problem is to check if there exists an assignment of the functions `x` and `y` that satisfies these assertions.

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ x (* 2 y)) 20))
(assert (= (- x y) 2))
(check-sat)
```

3.3 Inferring Collecting Semantics of Java Programs

We need to perform both intraprocedural and interprocedural analysis for analyzing deep logical properties of Java programs. In intraprocedural analysis, from a data flow analysis of the source code, in a series of steps we build a constraint system that captures its collecting semantics. In case of interprocedural analysis, we need to build a call graph and define some external rules relating the different API invocations and detect if the analyzed code breaks these rules.

3.3.1 Constraint System of Intraprocedural Analysis

In our analysis framework, we follow the following sequence of steps to check if the a program satisfies a user-defined analysis aspect.

1. We first perform a dataflow analysis of the Java source code and generate its collecting semantics

2. Based on the dataflow analysis results, we generate the static single assignment [43] graph of the program

3. We convert the SSA graph to the SMT-LIB formulas (see below)

4. Finally, we import models of uninterpreted API invocations as first order logic assertions

We illustrate the above steps using the following example:

```
class udhpcd{
  int getSocket(int listen_mode){
    int fd = 0;
    if (listen_mode == 2){
      fd = listen_socket();
    }
    else{
      fd = raw_socket();
        }
    }
}
```

In the program above, `listen_mode` is the user input. `listen_socket()` is a method, which will return a positive integer; `raw_socket()` is a method provided by operating system, which will return a specific integer number greater than zero.

First, we perform a data flow analysis of the source code. Figure 3.4 describes the results of dataflow analysis with the output constraints representing the collecting semantics of the program and Figure 3.5 represents the data flow of the program; the integer number in the data flow graph indicates the line number of each statement in the program.

At line 13 the value of the variable `fd` can be the return value of `listen_socket()` or `raw_socket()`; since this program has two branches. At compile time, we cannot determine which path the control will follow; so we consider the value of `fd` is $\{listen\_mode = 2 \bigwedge fd = listen\_socket(); listen\_mode \neq 2 \bigwedge fd = raw\_socket()\}$ where the semicolon represents disjunction.

To construct SMT-LIB logic formulas capturing the collecting semantics of the program, we need to convert the program to a static single assignment graph as in Figure 3.6.

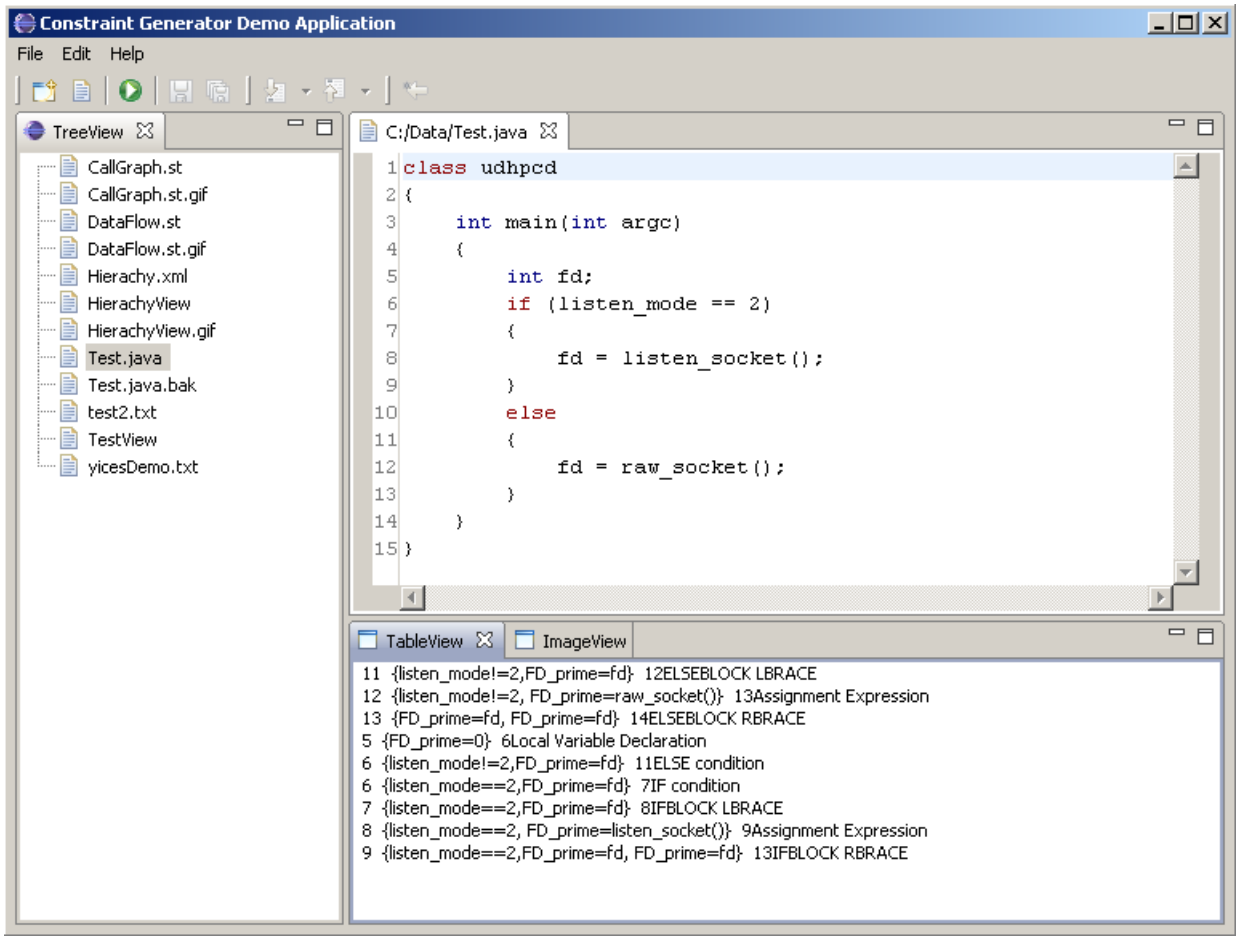

Figure 3.4: Collecting Semantics of Java Program

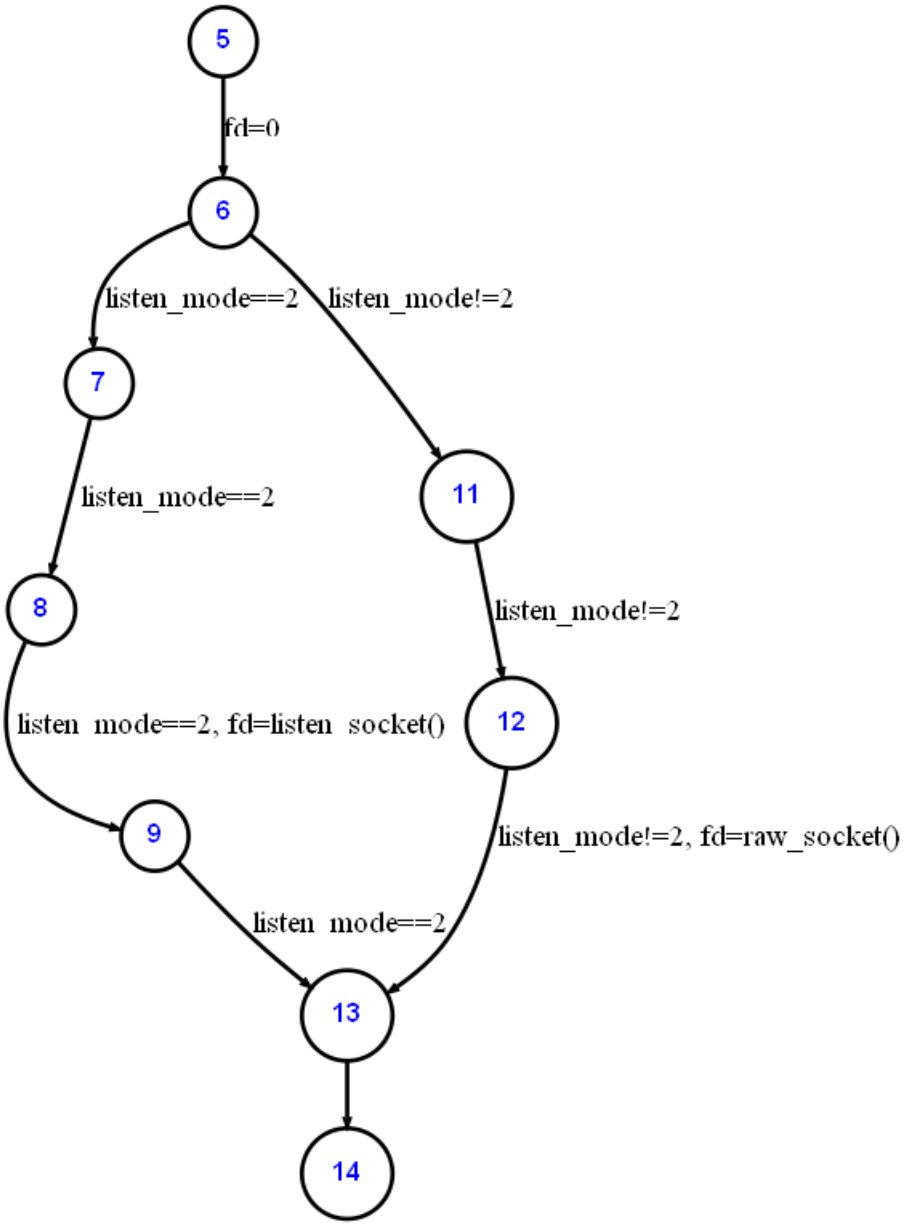

Figure 3.5: Dataflow of Java Program

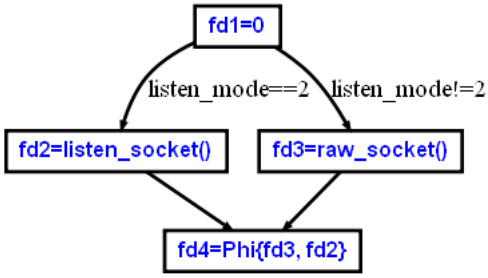

Figure 3.6: Static Single Assignment of Java Program

From the SSA graph in Figure 3.6, we create an assertion for each node. For example, the first node in the graph is fd1=0, we can create (assert (= fd1 0)). The node of "fd4=Phi{fd3,fd2 }" is a ϕ function, we build a disjunction (or (= fd4 fd3)(= fd4 fd2)). There are two labeled edges, so we need to create two implications: (=> (= listen_mode 2) (= fd2 listen_socket)) and (=> (distinct listen_mode 2) (= fd3 raw_socket)). The semantics of the APIs are incorporated as follows. If an API has no permission provided, we assert the API returns -1. Else we import an assertion characterizing the API from a model library.

For example, for the `listen_socket` and `raw_socket`, we import the assertion:(assert (and (>= listen_socket 0) (= raw_socket 1))).

The SMT-LIB formulas resulting from the SSA graph in Figure 3.6 is shown below.

```
(set-logic AUFLIA)
(declare-fun listen_mode () Int)
(declare-fun listen_socket() Int)
(declare-fun raw_socket() Int)
(declare-fun fd1() Int)
(declare-fun fd2() Int)
(declare-fun fd3() Int)
(declare-fun fd4() Int)
```

```
(assert (and (>= listen_socket 0) (= raw_socket 1)))
(assert (or (= fd4 fd3) (= fd4 fd2)))
(assert (= fd1 0))
(assert (and (=> (= listen_mode 2) (= fd2 listen_socket))
    (=> (distinct listen_mode 2) (= fd3 raw_socket))))
(assert (and(and(and (= listen_socket 3) (= raw_socket 1))
    (or (=> (= listen_mode 2) (= fd2 listen_socket))
    (=> (distinct listen_mode 2) (= fd3 raw_socket))))
    (or (= fd4 fd3)(= fd4 fd2))))
(check-sat)
```

The post condition of the program considered above is `fd`> 0. Verifying whether this post-condition holds is considered the analysis aspect for this program. This analysis aspect is incorporated into the SMT-LIB formula characterizing the collecting semantics of the program by adding the conjunct (<= fd4 0). The combined SMT-LIB formula was found to be unsatisfiable by the Yices solver indicating the program satisfies the specification.

We now describe an algorithm for converting an SSA graph of a program to SMT-LIB formulas that capture its collecting semantics. Let $\mathbb{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be the SSA graph of the program. In this graph, each node represents a statement in the program. We represent the if and loop conditions as edge labels in the graph. We can generate SMT-LIB formulas capturing collecting semantics of the program using algorithm 1 that formalizes the intuition described above.

Algorithm 1 Converting SSA to SMT Algorithm

for $n \in \mathcal{N}$ **do**
  if n is a simple assignment statement $VAR = EXP$ **then**
    Create an assertion (assert (= VAR EXP)) in SMT;
  end if
  if n is a assignment statement with API call $VAR = API$ **then**
    Create an assertion (assert (= VAR API));
  end if
  if n is a ϕ function statement **then**
    Let v be the variable in this statement and the set W be the values of this ϕ function;
    Create a disjunction $v = w_i$, where $w_i \in W$;
  end if
  if n is a function call statement $FUN()$ **then**
    Create an assertion (assert (= FUN FUN_SUMMARY));
  end if
  for $e \in \mathcal{E}$ **do**
    if e is labeled **then**
      Let n be the node directed by this edge;
      Create a conjunction of implication formula $e \rightarrow n$;
    end if
  end for
  if The API permission is provided **then**
    Provide the API model as (assert (= API API_specification value));
  else
    Set the API model as -1 (assert (= API -1);
  end if
  Provide the function summary as the function return value after the function analyzed (assert (= FUN_SUMMARY FUN_return value)).
end for

3.3.2 Call Graph and Logic Formulas of Interprocedural Analysis

We consider another example to illustrate the interprocedural analysis.

```
class camera{
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        preview = (SurfaceView) findViewById(R.id.preview);
        previewHolder = preview.getHolder();
        previewHolder.addCallback(surfaceCallback);
        previewHolder.setType(SurfaceHolder.
           SURFACE_TYPE_PUSH_BUFFERS);
    }
    private void takePicture(){
        camera.stopPreview();
        camera.takePicture(null, null, photoCallback);
    }
}
```

In this example, an Android application creates a `camera` object and uses this object to preview and take picture. The Android class documents [3] define clearly that "Important: Call startPreview() to start updating the preview surface. Preview must be started before you can take a picture." So we consider the safe property of this program is that the predicate `startPreview` must be true when the program needs to call the function `takePicture()`.

Figure 3.7 visualizes a call graph of this program; here we also include the line numbers of the function call statements. Our analysis tool automatically creates SMT-LIB formulas by defining predicates on the set of line numbers of the program. The predicates are defined as external rules in an XML file. We provide an user interface to help users define these rules. To construct these rules, users need to provide the predicate names and the tool needs to construct the relations.

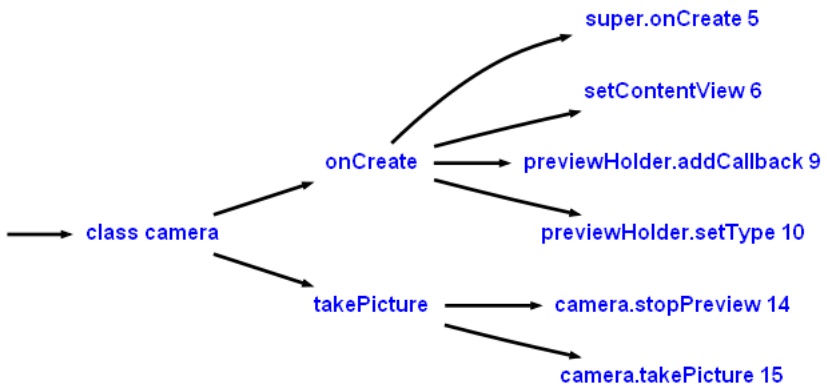

Figure 3.7: Function Call Graph of Interprocedural Analysis

In this example, we define predicates `startPreview`, `stopPreview`, and `takePicture`. At line number 15, where the function `takePicture()` is called, we consider the predicate `takePicture` to be true. So the formula `stopPreview` $\bigwedge$ `takePicture` holds true for line 15 since `stopPreview` is called in line 14. Our tool examines each program statement to set the appropriate values to these predicates; at line number where the function `stopPreview()` is called, so at line number 15, we have `stopPreview()` as `True`. The SMT-lib formulas can be constructed using Algorithm 1 as follows

```
(set-logic AUFLIA)
(declare-fun takePicture (Int) Bool)
(declare-fun stopPreview(Int) Bool)
(assert (exists ((x Int)) (=> (and (takePicture x) (not (
   stopPreview (+ x 1)))) (takePicture (+ x 1)))))
(assert (and (forall ((x Int)) (=> (stopPreview x) (stopPreview
   (+ x 1)))) (not (stopPreview 19))))
(assert (and (= x 14) (= (stopPreview x) true))
(check-sat)
```

The Android class documents [3] define clearly that "Important: Call startPreview() to start updating the preview surface. Preview must be started before you can take a picture." So we consider the safety property of this program to be the following: the predicate `stopPrivew` holds `false` when the program needs to call the function `takePicture()`. So we can construct the specification assertion $\mathcal{S}$ as (assert (and (= x 15) (and (takePicture x) (stopPreview x)))) assuming that the user inputs line 15 and `takePicture` as the test predicate and add a conjunct expressing its negation to the SMT-LIB formula above to incorporate the specification. The combined SMT-LIB formula was found to be satisfiable by the Yices solver. Analyzing the solution, we find that at line 15, the `stopPrivew` is called before `takePicture` violating the specification.

3.4 Experiments

In this section, we describe experiments that we conducted using our analysis framework.

We analyzed the source code from Android Bluetooth ChatServices application. This application builds a Bluetooth network platform to allow a device to exchange data with other Bluetooth devices. It has three main functionalities: 1. Discover the Bluetooth devices, 2. Paire and connect the devices, 3. Transfer data among these devices. The application uses several API calls such as: `BluetoothAdapter.getDefaultAdapter()`, `BluetoothSocket` and `BluetoothChatService`.

We provide these API models based on the Android Development Documentations. For this application, we simply consider that the application can set up the Bluetooth service and can connect to any devices discovered. So we set `BluetoothChatService` to return a `non-null` object, and in the SMT specification, we model the API function value as (not -1). The source

code analyzed satisfied the specification. We downloaded several free android application source code and ran our static analysis tool to the source code. In Table 3.2, we report some possible program vulnerabilities we detected from the analysis. In the application `Android SMSPopup`, we detected a possible command injection error; the command statement is an array that comes from another function which may possibly provide a wrong statement. In `openGPStracker`, we found that it has more permissions than required; this may lead to the overprivileged permission problem.

Table 3.2: Android Application Verification Experimental Results

Android SMSPopup [18]	1. In class SmsReceiverService.java there is a false null checker for statement. The branch "**if** (message.isSms()&&message. getMessageClass() ==MessageClass.CLASS_0)" will never be reached. 2. in class SmsPopupUtils.java the method getUnreadSmsCount(Context context) is never called. 3. in class SmsPopupUtils.java there is a system command call "Runtime.getRuntime().exec(commandLine.toArray (**new** String [0])) . getInputStream ())", this statement may lead to a command injection error.
NPR application [13]	1. The method execute () in Client.java does not perform any null checker before parsing XML, which gives an attacker the opportunity to supply malicious input. 2. The method constructList () in NewsTopicActivity.java does not perform any null checker for the list variable groupings before starting the loop, this problem may crash the program.
openGPStracker [14]	1. In class Constants.java on line 98, there is a hardcoded password. 2. The function serializeWaypoints () in GpxCreator.java fails to perform a null checker for variable mediaUri on line 440.
OpenSudoku [15]	1. The method saveToFile () in FileExportTask.java returns in a catch block on line 156, which may lead to a return value lost error.
Andar [2]	1. The method run () in CameraPreviewHandler.java calls a thread's run () method, but it should use start () method.
DaisyReader [6]	1. The method obtainEncodingStringFromFile () in ExtractXMLEncoding.java on line 74 fails to perform a null checker for the return value of readLine (), which might be null.

Chapter 4

Model-Based Static Code Analysis for MATLAB Models

MATLAB is widely used in scientific, engineering, and numerical computations. Complex systems such as digital signal processors, process control systems, etc. are modeled in MATLAB and analyzed; C implementation of the system can be automatically generated from the validated MATLAB model. We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework to analyze MATLAB code. The analyzer is generated by translating the collecting semantics of a MATLAB script to a formula in first order logic over multiple underlying theories. Function calls in a script can be handled by importing SMT assertions obtained by analyzing MATLAB files containing function definitions. Logical specification of the desired program behavior (rather its negation) is incorporated as a first order logic formula. An SMT-LIB formula solver treats the combined formula as a "constraint" and "solves" it. The "solved form" can be used to identify logical errors in the MATLAB model.

4.1 MATLAB Features

MATLAB is a dynamically typed language. A variable in MATLAB is considered as an array by default; so every value has some number of dimensions. Variables need not be declared, they can

accept any values that are assigned to them. The type of a numerical value in MATLAB is by default `double`. The built-in types of MATLAB can be summarized as follows:

- **double, sin**: floating point values;
- **int8, int16,int32,int64**: integer values;
- **logical**: boolean values;
- **char**: character values;

MATLAB functions are defined in `.m` files which have the same names as the functions. A function named `comp()` needs to be defined in a file with name `comp.m`. This file needs to be placed in the "current" directory or included in the MATLAB path. MATLAB functions can accept input arguments and output results in contrast with MATLAB scripts that can not accept any input nor generate outputs (other than printing on the workspace). MATLAB scripts are sequences of commands for simple computations and can invoke functions.

4.2 Verification Approach

Fig 4.1 describes the architecture of our verification approach. The abstract collecting semantics of a MATLAB script (or a function) is a represented as a first order logic constraint in the SMT-LIB syntax. This constraint will have "holes" or "markings" corresponding to invocation of functions in the script/function which need to get interpreted. Models of functions are created from collecting semantics of functions described in function files (represented as first order logic constraints in the SMT-LIB syntax). These models are used to unmark the abstract collecting

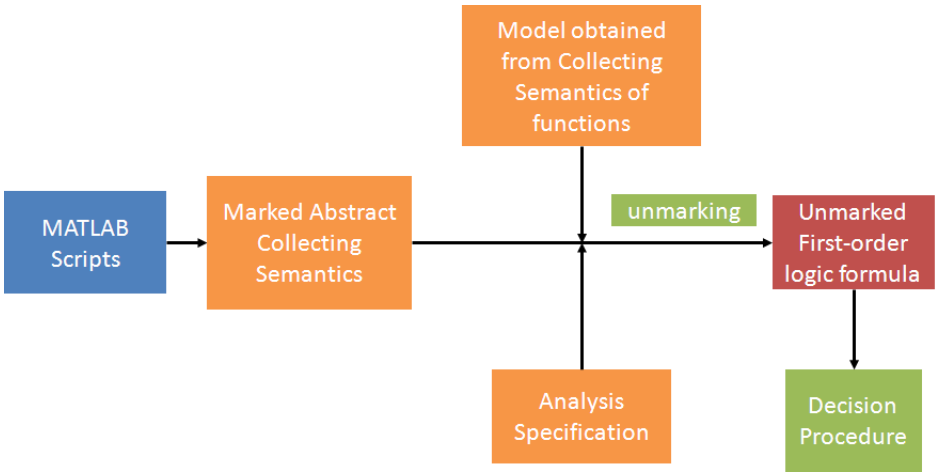

Figure 4.1: Architecture of Our Verification Framework for MATLAB Model

semantics by filling in the "holes". The negation of the property specification expressed as a formula in the SMT-LIB syntax is added to the combined constraints. The result is an "unmarked" first order logic formula that is presented to the decision procedure for satisfiability checking. We explain the steps in detail below. Let's consider the following example:

```
function s=comp(d)
advance = 0;
for x=1:50
    d = d+1;
    if d<50
        advance = 1/d;
    end
end

function bug()
    x = 10;
    comp(x);
    x = -4;
    comp(x);
```

In the example above, d is (the integer) is the formal parameter to the function comp() (call-by-value). It is incremented by 1 every time the loop executes. It is then used to determine the value of the variable `advance`. Checking whether the division operation at line 6 will cause a division-by-zero requires an interprocedural analysis to determine which values will be passed to the function comp().

In the code example, two values are passed to the function comp(). When called with `x=10`, d increases from 11 to 49. Line 6 will not result in a division by zero. However, when comp() is called with argument `x=-4`, d increases from -3 to 49. At some point, d will be equal to 0, causing a division by zero at line 6. A simple syntax check will not detect this run-time error.

We, first, generate a set of abstract constraints to describe the collecting semantics of the program (function or script), which overapproximates all the possible values for each variable. The constraints serve as an abstract intermediate representation of the code. Based on these constraints, we generate a dataflow graph of the program. The dataflow graph is used to generate SMT-LIB formulas describing the abstract collecting semantics. The dataflow graph of the function `comp` is described in Fig 4.2. In this figure, the integer number in each node is used to indicate the line number in the program.

From the dataflow graph, we create an assertion for each label. For example, the first node in the graph is advance=0, we can create (assert (= advance(0) 0)). This indicates that `advance` is initialized to `0`. The SMT-LIB formulas resulting from the dataflow graph in Figure 4.2 is shown below.

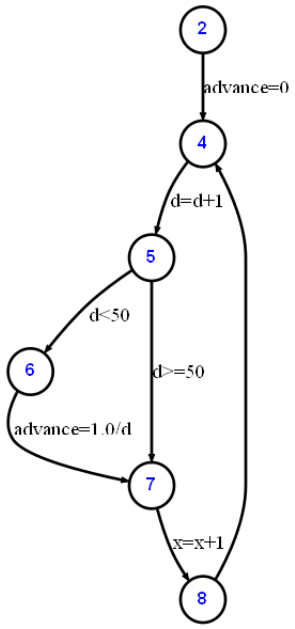

Figure 4.2: Dataflow of MATLAB Program

```
(set-logic AUFLIA)
(declare-fun advance () Int)
(declare-fun d () Int)
(assert (= advance(0) 0))
(assert (forall x Int) (=> (and (< x 50) (> x 2))
    (and (and (= d(1) (+ d(0) 1))
    (= d(x) (+ d(x-1) 1))))
    (and (=> (< d(1) 50) (= advance(1) (div 1 d(1))))
         (=> (< d(x) 50)(= advance(x) (div 1 d(x)))))))
(check-sat)
```

For the for-loop ranging from 1 through 50, we first describe the update of `d` and `advance` in the first iteration of the loop; here `d(0)` represents the initial value of `d`, i.e., the value with which the function comp is invoked. A universal quantifier over `x` with domain $[2, 50]$ is used to define the updates of `d` and `advance` during the second through the fiftieth iteration of the loop.

To detect if the program has a divide-by-zero error, we need to check if d can become zero within the for loop; this is the analysis aspect for this program. This analysis aspect is incorporated into the SMT-LIB formula characterizing the collecting semantics of the program by adding the conjunct (assert (exists x Int) (and (and (<= x 50) (>= x 1)) (= d(x) 0))) . The combined SMT-LIB formula was found to be satisfiable by the Yices solver indicating the program can have a division by zero error. While Yices is incomplete for quantified formulas, in most practical cases it was able to come up with a proof.

Let's consider another example in MATLAB:

```
if edgestop==1
    k=K;
    a=1;
elseif edgestop==2
    k=K*(2^0.5);
    a=1/(2*exp(-0.5));
elseif edgestop=='tky'
    k=K*5^0.5;
    a=25/32;
end
Gn=[I(1,:,:);I(1:row-1,:,:)]-I;
Gs=[I(2:row,:,:);I(row,:,:)]-I;
Ge=[I(:,2:col,:) I(:,col,:)]-I;
Gw=[I(:,1,:) I(:,1:col-1,:)]-I;
if edgestop==1
   Cn=1./(1+(Gn/k).^2).*a;
   Cs=1./(1+(Gs/k).^2).*a;
   Ce=1./(1+(Ge/k).^2).*a;
   Cw=1./(1+(Gw/k).^2).*a;
elseif edgestop==2
   Cn=exp(-(Gn/K).^2).*a;
   Cs=exp(-(Gs/k).^2).*a;
   Ce=exp(-(Ge/k).^2).*a;
   Cw=exp(-(Gw/k).^2).*a;
end
```

In this example, there is a typical mistake that almost all the developers make. In line 22, the statement should be Cn=exp(−(Gn/k).^2).∗a;; but in the program, the developer typed in the wrong variable name K. This error cannot be detected by the compiler; no error is reported at runtime either; but the program will simply spit out wrong results. Our approach can detect this problem, since we need to generate constraints to overapproximate all the possible values of each variable. In this example, if the variable edgestop is 1, the value of k is K∗(2^0.5), and the value of variable Cn is exp(−(Gn/k).^2).∗a. We can build the constraints as (edgestop=1) ⇒ (k=K ∗(2^0.5)) ⋀ (Cn=exp(−(Gn/k).^2).∗a). To verify the correctness of this MATLAB code, we need to set the post condition. Since the value K∗(2^0.5) > K, we can set the condition Cn<exp(−(Gn /15).^2).∗a; as the post condition to detect this variable misuse error. If the solver returns `sat` for the formula above, the program has a variable misuse error. The SMT formulas are followings:

```
(set-logic AUFLIA)
(declare-fun edgestop () Int)
(declare-fun k () Int)
(declare-fun K () Int)
(declare-fun Cn () Int)
(declare-fun Gn () Int)
(declare-fun a () Int)
(declare-fun sqrt(Int))
(declare-fun pow(Real Int) Real)
(declare-fun exp (Real) Real)
(define-fun div ((x Real) (y Real)) Real
  (if (not (= y 0.0))
      (/ x y)
      0.0))
(assert (= K 15))
(assert (= edgestop 1))
(assert (=> (= edgestop 2) (= k (* K (sqrt(2))))))
(assert (=> (= edgestop 2) (= Cn exp(* (pow (div Gn k) 2) a))))
(assert (=> (= edgestop 2) (>= Cn exp(* (pow (div Gn K) 2) a))))
(check-sat)
```

We now describe an algorithm for converting an dataflow graph of a program to SMT-LIB formulas that capture its collecting semantics. Let $\mathbb{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be the dataflow graph of the program. In this graph, each node represents a statement in the program. We represent the if and loop conditions as edge labels in the graph. We can generate SMT-LIB formulas capturing collecting semantics of the program using algorithm 2 that formalizes the intuition described above.

In this algorithm, we first visit each label in the dataflow graph to declare functions in SMT for all variables that are used in the code. Then, we visit each node to detect if there are any nodes that have children which have smaller line number than itself; such a node indicates a loop in the code. Assume that the conditions on this loop are given by $expr1 : expr2$. For each assignment statement inside this loop, we translate the statement as follows. Let d be the variable in the left side of the assignment statement with $assexpr$ on the right side. We create an assertion (assert (= d(exp1) assexpr(0))) as the base condition to indicate the update of d the first time the loop is executed where assexpr(0) is obtained from assexpr by replacing each variable x occurring in it by x(0), and create an assertion (assert forall (i Int) (=¿ (and (>= i exp1) (<= i exp2) (= d(i) assexpr(i)/(i-1)) where assepr(i)/(i-1) represents replacing each variable x by x(i) if x has been updated in a predecessor node in the loop else by x(i-1) (obtained from use-define links) for each statement inside the loop body to express the updates in the remaining executions of the loop. If there is no loop, we can simply create assertion (assert (= VAR EXP)). To convert the `if/else` block in the code, we create a conjunction of implication formula.

Algorithm 2 Converting MATLAB Program to SMT Algorithm

for $e \in \mathcal{E}$ **do**
  if e is a simple assignment statement $VAR = EXP$ **then**
    Create a definition (define-fun VAR (EXP)) in SMT;
  end if
  if e is a function call statement $FUN()$ **then**
    Create an assertion (assert (= FUN FUN_SUMMARY));
  end if
  for $n \in \mathcal{N}$ **do**
    if n has two children **then**
      Create a conjunction of implication formula $e \rightarrow$ the children labels;
    end if
    if n has child whose line number is less than n **then**
      Evaluate the expression of the label from n to it's child;
      Let $exp1$ be the initial condition, $exp2$ be the end condition;
      for all the labels from the child of n to n **do**
        Let d be the variable in the left side of the assignment statement with $assexpr$ on the right side;
        Create an assertion (assert (= d(exp1) assexpr(0))) where assexpr(0) is obtained from assexpr by replacing each variable x occurring in it by x(0);
        Create an assertion (assert forall (i Int) (=¿ (and (>= i exp1) (<= i exp2)
        for all the labels from the child of n to n **do**
          (= d(i) assexpr(i)/(i-1)) where assepr(i)/(i-1) represents replacing each variable x by x(i) if x has been updated in a predecessor node in the loop else by x(i-1) (obtained from use-define links)
        end for
      end for
    end if
  end for
  Provide the function summary as the function return value after the function analyzed (assert (= FUN_SUMMARY FUN_return value)).
end for

4.3 Experimental Results

We implemented our analysis framework in Java (using ANTLR) with Yices as SMT solver. We analyzed the MATLAB examples including matrix computation and signal processing obtained from [1]. Many of the examples were found to meet their specifications. However in several examples, our analysis framework found logical errors. These results are summarized in Table 4.1. All experiments were run on desktop with a Pentium dual-core CPU 2.6 GHz running Windows XP. The time needed for verification was never more than a minute.

Table 4.1: MATLAB Verification Experimental Results

GPC_timu.m	Line 113: The "if ite1 < N" block is never reached. Line 151: the statement may have division by zero error; since "k-tao-i2" can be zero.
GAconstrain.m	Line 63: The statement "if nmutationR>0" is not valid, since nmutationR is always larger than 0.
GA.m	Line 186: The if statement "if maxvalueRAND(m-m0)<maxvalueRAND(m-(m0+1))" is not valid, since m-m0 is greater than m-(m0+1). The random return value may make this condition always false.
PSK_carrier_timing_est.m	Line 178: the matrix index may be out of bound. Line 187: the statement "nco_l(k)=exp(-j*(2*pi*f0*Ts(start_diff+n+round(Kc2*err_tao(k))-8) +Kc*Uc(k)));" is not valid. It should use "fe" not "f0".
Felics.m	Line 63: the matrix index may be out of bound.
Kalman filtering.m	Line 63: The parameters in function "lmodeinitial(T,r,zx,zy,vxks,vyks,perr2)" are invalid.
emd.m	Line 159: the matrix index may not match. Line 352: the statement may have division by zero error.
TV_denoise.m	Line 37: The loop "while(i<niter)" may never terminate, since the value of variable "iflamda" is not assigned, the statement "i=i+1" may never be reached. Line 53: the statement may have division by zero error.
smooth_diffusion.m	Line 79: The statement uses a wrong parameter "K". Line 105: the function "imshow" has an invalid parameter "uint8()", since the function "uint8()" needs input.

Chapter 5

Formal Verification of Commercial Wireless Router Firmware

Formal verification of the trusted computing base of a software system is essential for its deployment in mission-critical environments. Commercial-of-the-shelf routers are nowadays being used for managing traffic in high-assurance networks. The specifications for the development of these routers are provided by RFCs that are only described informally in English. It is essential to ensure that a router firmware conforms to its corresponding RFC before it can be deployed for managing mission-critical networks.

We report the formal verification of the conformance of the open source Netgear WNR3500L wireless router firmware implementation to the RFC 2131 [24] based on which it is designed. The formal verification effort led to the discovery of several possible problems in the implementation that we report in this paper. We have used the Coq proof assistant extensively in this verification effort. The formal verification process demonstrates the usefulness of inductive types and higher-order logic in software certification.

5.1 Coq Preliminaries

We first provide a brief introduction to the Coq proof assistant. We start with the definition of inductive type. For example: `nat`, the simplest inductive type can be defined as

```
Inductive nat : Set :=
      O : nat
    | S : nat -> nat.
```

Here, `O` and `S` are called *constructors*; `O` stands for the natural number zero and `S` is the successor function. We can also define an inductive type by enumerating its elements.

```
Inductive month : Type :=
Jan | Feb | Mar | Apr | May | Jun
| Jul | Aug | Sep | Oct | Nov | Dec.

Inductive season : Type :=
Spring | Summer | Fall | Winter
```

The various names Jan, Feb, etc., are *constructors*. After we have defined this inductive type, we can compute values according to which element it is matching. For example:

```
Definition getSeason (m:month) :=
match m with
Apr => Spring | Jun => Summer
| Sep => Fall | Nov => Winter
end.
```

We can define recursive functions using the type `nat`. For example, the sum of two natural numbers can be defined as follows [4].

```
Fixpoint plus (n m : nat)
    {struct n}:nat
  := match n with
    O => m
    | S p => S (plus p m)
end.
```

where `{struct n}` is used to denote that successive recursive calls of the function `plus` are invoked with decreasing values of `n`.

Coq has the type `Prop` for propositions. We can define a predicate `P: nat ->Prop` for all natural numbers using the type `Prop`. Assume that we have already provided a proof of `P 0` and a proof for `forall n, P n -> P (S n)` that we call `nextstep`, we can construct a proof of `P n` for all natural numbers n, using the following functional program:

```
Parameter P:nat->Prop.
Parameter nextstep: forall n,
    P n -> P (S n).
Fixpoint natind (n:nat)
  {struct n}: P n
  := match n return (P n) with
  | 0 => P 0
  | S q => nextstep q (natind q)
end.
```

The type of the recursive function `natind` is `forall n, P n`, which is a dependent type, since the type of this result depends on the value of the argument. `nextstep` is a function takes natural number `n` and proof of `P n` as arguments returning a proof of `P (S n)` [45].

5.2 Formal Verification of Router Firmware

The Netgear WNR3500L router is the first product, encouraged in industry, for which customers can choose between the manufacturer's router firmware and several open source alternatives.

Netgear has come to recognize that there are many customers who transform low-cost router equipment into high-end network devices by using advanced firmware. They openly support and encourage people to publish their updated firmware [12]. It is essential to ensure that a router firmware contributed by the open source community conforms to its corresponding RFC before it can be deployed for managing mission-critical networks.

5.2.1 Verification Strategy

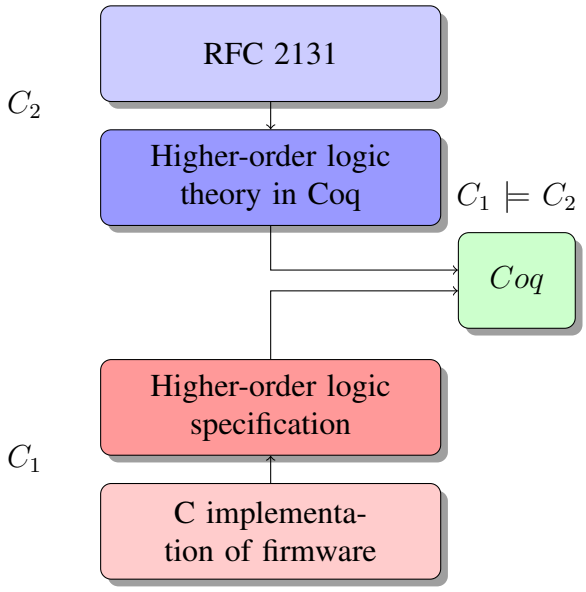

Figure 5.1: The Higher-order Logic Refinement

Figure 5.1 describes our verification strategy. The RFC 2131 defining the implementation of the firmware for the Netgear WNR3500L router is formalized as a theory in Coq. We call this specification C_2. Similarly, a Coq model for the C code implementing the firmware is developed. We call this C_1. The verification effort intends to show that C_1 refines C_2, i.e., it intends to prove the following theorem. Verification is carried out in Coq by showing $C_1 \models C_2$.

***THEOREM* 1.** *The firmware conforms to RFC 2131 iff* C_1 *refines* C_2 *iff* $C_1 \models C_2$

5.3 Formalization of the RFC

We first show the formalization of the RFC 2131 in Coq. For illustrative purposes, we show the formalization of the DHCP protocol as described in the RFC through informal English and sequence diagrams in Coq. The rest of the Coq specification for the RFC can be downloaded from the website listed in the Introduction.

5.3.1 DHCP Protocol

DHCP contains two mechanisms: a protocol for delivering configuration parameters from a DHCP server to a client and a mechanism for allocation of network addresses to clients. For the allocation of a network address, DHCP supports three mechanisms:

- Automatic allocation: an IP address is assigned permanently.
- Manual allocation: a client's IP address is assigned by the user and DHCP is used simply to confirm the assigned address to the client.

- Dynamic allocation: a client requests a free address for some period of time. The DHCP server needs to provide the address and guarantees not to reallocate that address within the requested time and returns the same network address each time the client requests an address.

The dynamic allocation is the most widely used one.

5.3.2 DHCP Client-Server Interaction

We give a description of the DHCP message exchange between clients and servers [24]. Figure 5.2 shows the exchange of messages between the client and the server according to the RFC.

- The client broadcasts a `DHCPDISCOVER` message on its local physical subnet. The `DHCPDISCOVER` message may include options that suggest values for the network address and lease duration.

- Each server may respond with a `DHCPOFFER` message that includes an available network address in the `yiaddr` field. When allocating a new address, servers should check that the offered network address is not already in use.

- The client receives one or more `DHCPOFFER` messages from one or more servers. The client chooses one server from which to request configuration parameters, based on the configuration parameters offered in the `DHCPOFFER` messages.The client broadcasts a `DHCPREQUEST` message that includes the 'server identifier' option to indicate which server it has selected.

- The servers receive the DHCPREQUEST broadcast from the client. Those servers not selected by the DHCPREQUEST message use the message as notification that the client has declined that server's offer. The server selected in the DHCPREQUEST message commits the binding for the client to persistent storage and responds with a DHCPACK message containing the configuration parameters for the requesting client.

- The client receives the DHCPACK message with configuration parameters and the duration of the lease specified in the DHCPACK message. After waiting for a minimum of ten seconds the client can start the configuration process.

5.3.3 The DHCP Protocol Specification in Coq

A DHCP client first needs to initialize, that includes creating a random xid used to communicate with a DHCP server, configure the host name if users input some specific name, and retrieve the process identification number (pid) from the pidfile. The communication messages can be enumerated as an inductive type:

```
Inductive DHCP_MESSAGE :=
  DHCPDISCOVER|DHCPREQUEST|DHCPDECLINE
  |DHCPRELEASE|DHCPACK|DHCPNAK|DHCPINFORM.
Inductive arg_options:= c|H|p.
Parameter char:arg_options.Parameter random_xid :Set.
Parameter configure_host : arg_options->Set.
Parameter configure_pid:arg_options->Set.
Definition clientInit (op:arg_options) :Set :=
  match op with
  c => random_xid
| H => configure_host H
| p => configure_pid p
end.
```

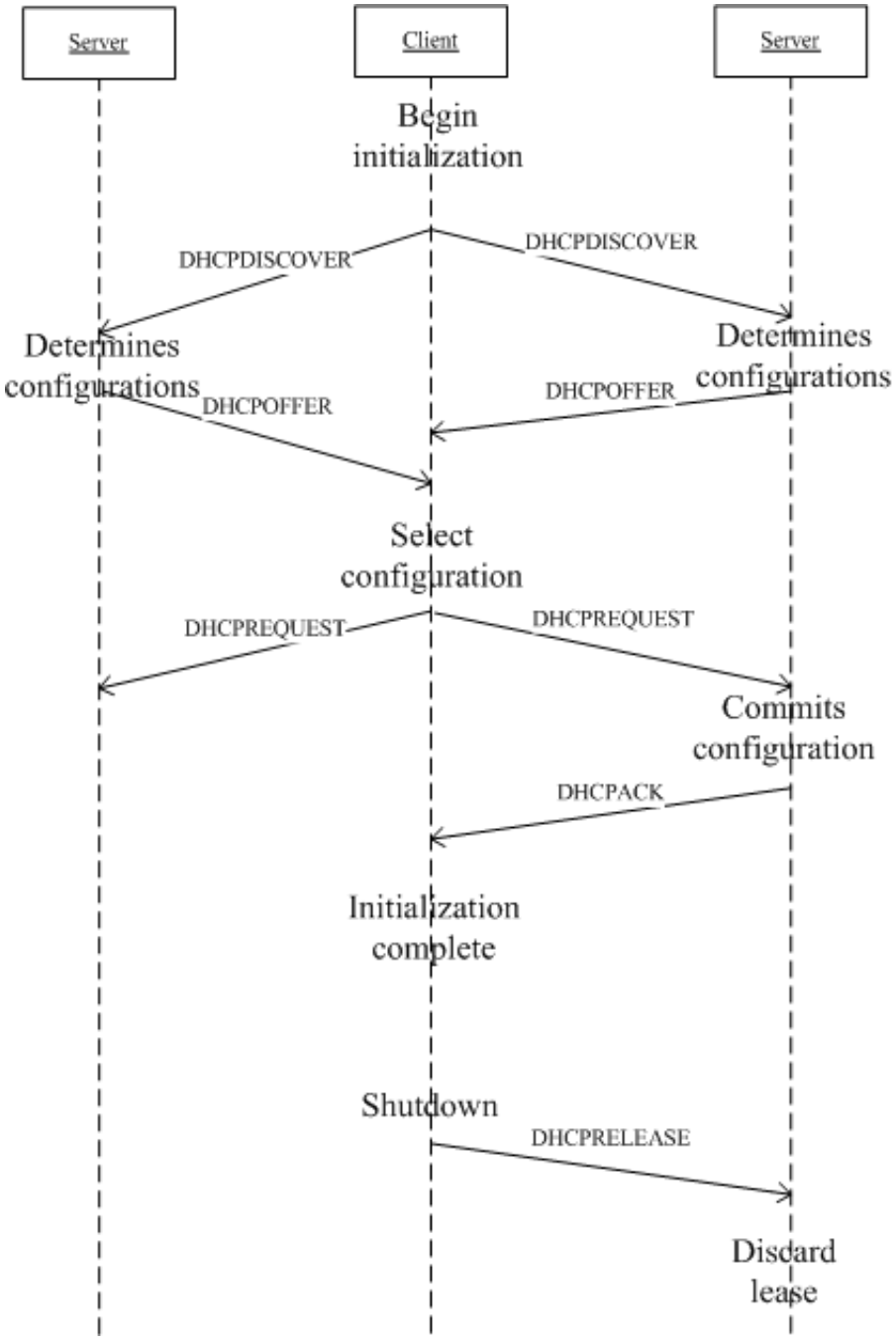

Figure 5.2: Messages Exchanged Between DHCP Client and Servers

After the initialization, the client will send `DHCPDISCOVER` and wait for replies from servers; we formalize this as follows:

```
Parameter initPacket : DHCP_MESSAGE -> dhcpMessage.
Parameter addSimpleOption: (dhcpMessage)->nat-> dhcpMessage.
Definition sendDiscover (xid:nat) (requested_ip:nat) :=
 let packet := initPacket DHCPDISCOVER in
    addSimpleOption packet requested_ip.
```

According to the RFC specification, the DHCP server first needs to create a socket, and then use this socket to get packets sent by the DHCP clients; this is formalized as:

```
Parameter listen_socket:nat->nat->string->nat.
Theorem server_socket:
  listen_socket ip port server_config > 0.
Theorem get_packet:
  getPacketSize packet > 0 -> server_socket.
```

After receiving a valid packet, the DHCP server needs to allocate IP and setup lease. It needs to check if the requested client has already setup a lease; if not, it will find a new free IP address to assign to the client. If, unfortunately, there is no free IP address to allocate, it needs to check if it can find any expired lease to free the corresponding IP address. These properties are formalized by the following Coq specification.

```
Parameter addr:nat. (* IP address for allocation *)
Parameter chaddr:nat. (* client hardware address*)
Parameter expiration:nat. (* The time duration for expiration*)
Parameter leases:list nat. (* The lease table*)
Parameter find_lease_by_chaddr :nat -> list nat -> nat.
Definition server_config_end := 255.
Definition server_config_start := 0.
Theorem findFreeIP:
  forall freeaddr:nat,
  freeaddr>server_config_start /\ freeaddr<server_config_end /\
     find_lease_by_chaddr freeaddr leases=0.
```

```
Parameter findExpiredIP :nat-> nat -> nat.
Theorem findExpiredLeaseEntry:
  findExpiredIP leases expiration >0.
Theorem allocationIP:
  find_lease_by_chaddr chaddr leases >0 \/ findExpiredIP leases
     expiration >0.
```

The above formalizations can be regarded as properties that should hold for a correct firmware implementation following the RFC.

5.3.4 Formal Modeling of the C Implementation of the Firmware

To verify the conformance of the C implementation of the firmware with RFC 2131, we need to formalize it in Coq. In [59], the authors provide simplified description of the protocol at an abstract level; however, in our paper, we focus on the implementation details with assumptions that there are no failures from the operating systems side (i.e., we assume that sockets can be successfully created, etc.). For example, when we attempt to model the implementation of how the DHCP server allocates IP addresses to clients, when the it receives `DHCPDISCOVER` messages, we assume that the server receives the message correctly, i.e., there is no message loss during the communication between the clients and the server.

In the following, codelets from the C implementation of the firmware are described using `verbatim` fonts (including C functions, variables, and references to them in the text) while Coq specifications are described using sans−serif ML−Style fonts (including variables, functions, formulas, and references to them within the text). We illustrate the specification of the C implementation assuming only one DHCP server in the network. In this case, the WNR3500L wireless router is required to respond to every `DHCPDISCOVER` messages from clients with

a DHCPOFFER message including an available network address. In the source code of router firmware, the IP address allocation process is implemented by the following steps:

1. The server checks if the requested client is previously allocated by calling the function `find_lease_by_chaddr(chaddr)`. If the return value is not null, the server uses this returned IP address.

2. If the return value is not null but it is expired, the server needs to check if this IP address is available by calling function `check_ip(u_int32_t addr)`. In WNR3500L wireless router, it simply sends an arp packet to check if the IP address is in use.

3. If the client is not previously allocated, the server will select the first free IP address in the range of `server_config.start` to `server_config.end` which are defined in the file `/etc/udhcpd.conf`. If the server can find a free IP address, then it allocates this free IP address.

4. If there is no free IP address, the server needs to check already allocated IP addresses which are expired. If the server can find an expired IP address, it needs to check if it is still in use.

After allocating an IP address, the server needs to set up the lease for this IP address. The server must choose an expiration time for the lease as follows:

1. If the client already has an assigned IP address, the server returns the lease expiration time previously assigned to that address unless the client explicitly requests a specific lease to extend the expiration time.

2. If the client has no assigned IP address, the server assigns a locally configured default lease time.

In the WNR3500L router source code, the DHCP server maintains an array `leases`, which has type `struct dhcpOfferedAddr` (C `struct` as opposed to the Coq keyword), to keep track of IP address allocation information. Each entry in this array contains the client hardware address, the allocated IP address, and the expiration time. The data structure `dhcpOfferedAddr` can be modeled as a record.

```
Record dhcpOfferedAddr := mkdhcpOfferedAddr
{
  chaddr: list nat;
  yiaddr: nat;
  expires: nat;
  hostname: string
}.
```

After receiving the `DHCPDISCOVER` message, the steps for the DHCP server include checking the `leases` table and finding a free IP address. This can be specified in Coq as:

```
Fixpoint find_lease_by_chaddr (chaddr:nat) (leases:list nat):nat
    :=
  match leases with
    | nil => 0
    | a::h => match eq_nat_dec chaddr a with
              | left _ => S (find_lease_by_chaddr chaddr h)
              | right _ => find_lease_by_chaddr chaddr h
              end
  end.
Definition findLease:=
    find_lease_by_chaddr chaddr leases = 1.
Theorem findLeaseEntry:
  find_lease_by_chaddr chaddr leases > 0.
Proof.
 unfold findLease;trivial.
Qed.
Fixpoint inLease (addr:nat) (leases:list nat) :nat :=
  match leases with
  | nil => 0
```

```
  | (a::h) => match a with
              | addr => 1 + inLease addr h
              end
  end.
Definition server_config_end:=255.
Fixpoint findFreeIP (address:nat) (leases:list nat):=
  match address with
  0 => 0
  | server_config_end => 0
  | S p => inLease p leases + (findFreeIP p leases)
  end.
```

If the server cannot find any available IP address to allocate, it needs to choose an expired lease and free the corresponding IP address. The WNR3500L DHCP server simply compares the expiration value from the `leases` table with the current time and finds the oldest lease to replace with a new lease. This is formalized in Coq as follows.

```
Fixpoint max (n m:nat){struct m}:nat :=
  match n, m with
  | 0, _ => m
  | S n', 0 => S n'
  | S n', S m' => S (max n' m')
  end.
Fixpoint oldest (expiration:nat) (leases_expiration:list nat) {
   struct lease_expiration} :nat :=
  match leases_expiration with
  | nil => expiration
  | a::h => match (max expiration a) with
            | expiration => oldest expiration h
            end
  end.
Fixpoint findExpiredIP (addr:nat) (leases:list nat) (expiration:
   nat) {struct addr} :nat :=
  match addr with
  | 0 => 0
  | S p => (oldest expiration leases) + (findExpiredIP p leases
     expiration) end.
```

To show that the DHCP server can successfully assign IP addresses to clients sending DHCPDISCOVER, we need to prove the following theorem. It states that after all the IP checking steps are completed, the DHCP server will be able to find a valid available IP address (a valid IP address is modeled as a positive one).

```
Theorem allocationIP:
  find_lease_by_chaddr chaddr leases > 0
  \/ (findFreeIP server_config_end leases)>0
  \/ (findExpiredIP addr leases expiration)>0.
```

5.4 Deductive Verification of Router Firmware

To verify the conformance of the router firmware to the corresponding RFC we prove that the Coq specification for the implementation entails that for the RFC. In this section, we show how to verify that the WNR3500L DHCP server successfully responds to a DHCPDISCOVER message with a DHCPOFFER message. The rest of the proofs are available from the website listed in the Introduction. To prove that the WNR3500L DHCP server successfully responds to a DHCPDISCOVER message with a DHCPOFFER message, we need to verify that the source code satisfies following rules:

1. The server has an available IP address to allocate.

2. The server can setup lease for this IP address.

3. The IP address is not currently used in the network.

5.4.1 Detection of Deviation from RFC

To verify the correctness of adding a new lease into the `leases` array, we need to prove that if the server has no free IP address to assign to its clients, it must always find an expired lease to replace the new lease (according to the RFC). However, we fail to obtain such a proof. This is because the server only keeps the expiration time value and compares it with the current time but does not update the `leases` table before allocating an available IP address. In our model, the function (oldest expiration leases) will always return a value which is less than expiration , which means that the formula findExpiredIP is always false. When the server has no free IP address to allocate, it cannot find any expired lease so as to free the corresponding IP address. This problem will cause the server to fail to send the `DHCPOFFER` message resulting in clients failing to get IP addresses in the network, although the server should have been able to obtain available IP addresses to allocate. The reason is that the C function `sendOffer` does not conform to the RFC specification (i.e., the Coq specification "allocation" above). It fails to check the availability of an IP address; it only checks the `leases` table and fails to replace the appropriate expired lease. In a home wireless network setup, this problem leads to a "limited connection" error.

To fix this problem, we suggest that the server broadcasts an ARP packet [76] to clients for all the allocated IP addresses in the `leases` list and waits for valid ARP replies until time out. If the return value is "0", it means that the particular IP address is occupied by a client and this client responds correctly; the server cannot use this IP address. Otherwise, if the return value is "1" or "-1", it means that the client is not available or that some errors have occurred; the server can free this IP address and assign it to other requested clients. The server of course needs to first initialize the return value `rv` to "1". If the clients respond to the above ARP message correctly,

the return value will be “0”. In the following, we will show how to construct a program to set up this ARP packet. Then we prove that under the above scheme, the return value to the server will be “0”.

We first build models for `sockaddr`, `arpMsg` and `timeval` as inductive types:

```
Parameter char : Set.
Parameter time_t : Set.
Inductive ilist : nat -> Set :=
  | Nil : ilist O
  | Cons : forall n,
    char -> ilist n -> ilist (S n).

Inductive structType:Type :=
  arpMsg:Z->Z->Z->Z->Z->char->
    char->char->char->char->structType
  | sockaddr:forall n, (ilist n)->structType
```

We define an inductive type ilist to model the array type in the C program; ilist is a dependent type, since the return result depends on the match of n. Then, we define an inductive type structType and use sockaddr, arpMsg as constructors.

The constructor arpMsg takes `htype` (hardware type,must be `ARPHRD_ETHER`), `ptype` (protocol type, must be `ETH_P_IP`), `hlen` (hardware address length, must be 6), `plen` (protocol address length, must be 4), `operation` (ARP operation code), `sHaddr` (sender’s hardware address, which is a char array with length 6), `sInaddr` (sender’s IP address, a char array with length 4), `tHaddr` (target’s hardware address), `tInaddr` (target’s IP address), `pad` (Ethernet payload) and returns the `structType` type. The function `sockaddr` takes an array of address data as arguments. The function `timeval` takes `tv_sec` which has type `_time_t` provided by the operating system (Linux).

After these declarations, the ARP program initiates a `socket`. Since the DHCP server needs

this ARP program to return "0" to continue to the next lease offer step, the function `socket` must return a positive value. We assume an axiom that states that the socket creation function will return a correct value since this function is provided by the operating system.

```
Definition socket:Set->Set->Z.
Theorem createSocketSuccess :
 forall createsocket:Z, forall t:Set, forall protocol:Set,
 forall (HW: createsocket = (socket domain t  protocol)),
  (createsocket > 0).
```

After successfully creating a socket, the ARP program needs to create an ARP packet and set appropriate protocol arguments for it by the following steps: 1. allocates memory, 2. sets protocol type as `htons(ETH_P_ARP)`, 3. sets the destination MAC address as broadcast. Then it calls the function `sendto()` to broadcast this ARP packet. Again this function needs to return a positive value; otherwise, the return value for the ARP program will be set to "0". In Coq, we can define this function and build the Coq proof obligations as follows.

```
Definition sendto (s:Z) (arp:structType)
 (size:Z) (a:Z)
 (addr: structType)
 (size:Z) :=
  match arp with
  arpMsg htype ptype  hlen plen
    operation mac ip yiaddr =>
     match addr with sockaddr interface => 1
            | _ => 0
         end
       | _ => 0
       end .
Theorem sendToSuccess:
  forall s :Z, forall arp:structType, forall addr:structType,
 (sendto s arp 0 0 addr (sizeof addr)) > 0.
```

Here we define the function sendto that only checks if the program provides the appro-

priate arguments for arpMsg and sockaddr. We assume there are no computational errors for the arguments accepted by arpMsg; this assumption is based on the fact that variables such as `ARPHRD_ETHER` are defined by the operating system as is the function `htons` which converts the unsigned short integer from host byte order to network byte order.

For the definition of the function sendto, we need to show that this function will return a positive number. As we have explained previously, we only need to check the program arguments; all variables and functions are provided by the operating system and so we can directly assume this statement as an axiom.

After broadcasting the ARP packet, the program will call the function `select()` to discover which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, `select()` blocks, up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors or until a signal arrives that needs to be delivered. To accomplish this logic, the program needs to initiate the declared file descriptor `fdset` to have zero bits and put it into the system file descriptor set. The Coq definition and obligation rules are modeled as follows. We will assume the formula selectSuccess as an axiom since the function `select` is supported by the operating system.

```
Fixpoint FD_ZERO (size:nat) :=
  match size with 0 => nil
  | S p => 0::(FD_ZERO p)
end.

Fixpoint FD_SET (s:nat) (fdset:list nat) :=
  match s with 0 => fdset
  | S p => match fdset with nil => s::fdset
           | h::t => s::(FD_SET p t) end end.
```

```
Definition select (s:Z)(fdsetRead: list Z)(fdsetWrite: list Z)
    (exception:list Z)
    (timeVal:nat) :Z.
Theorem selectSuccess :
 forall s:Z, forall fdsetRead: list Z, forall fdsetWrite: list Z
    ,
 forall timeval:Z,
 select s fdsetRead nil nil timeval > 0.
```

In next step, the server needs to call the function `recv()` to receive the replies after broadcasting the ARP packet. We can define the function `recv()` as:

```
Definition recv (s:Z) (arp:structType)
(size:Z) (a:Z) :=
  match arp with
  arpMsg htype ptype  hlen plen
  operation mac ip yiaddr =>1
       | _ => 0
       end .
Definition arppacket:= arpMsg htype ptype  hlen plen operation
   mac ip yiaddr.
Parameter s:Z.
Theorem recvSuccess:
  forall size:Z,
  (recv s arppacket size 0) > 0.
Proof.
intros.
unfold recv; unfold arppacket.
omega.
Qed.
```

We can conclude from the previous discussion that for the value returned by the ARP program to be "0", the program needs to create a socket successfully, call the `sendto` function correctly, discover an available file descriptor, and receive correct replies from clients before time out. We will show that the Coq specification of the ARP program will entail the following predicate and prove that the following theorem holds true for the ARP program.

```
Parameter arp:structType.
Parameter addr:structType.
Parameter size:Z.
Parameter fdsetRead: list Z.
Definition timeval := 2%Z. (*set the time out as 2 seconds*)
Theorem arpping:
  forall rv:Z,
  (select s fdsetRead nil nil timeval > 0)
  /\ (sendto s arp 0 0 addr 128 >0 )
  /\ (recv s arppacket size 0 > 0).
Proof.
  intros.
  split; apply (selectSuccess s fdsetRead nil timeval).
  split; apply (sendToSuccess s arp addr).
  apply (recvSuccess size ).
Qed.
```

Based on the construction of the ARP program, we can update the Coq specification of the C implementation as follows,

```
Definition arpping (yiaddr:nat) := 0.
Definition checkAddr (addr:nat) := (arpping addr).
```

and the following theorem holds for the modified specification.

```
Theorem allocationIP:
  (findLease yiaddr leases chaddr) > 0
  \/ (findFreeIP server_config_end leases)>0
  \/ (findExpiredIP addr leases expiration)>0
  \/ ((findExpiredIP addr leases expiration)<=0 /\checkAddr addr
      = 0)-> addr>0.
```

In this specification, we can see that even if (findExpiredIP addr leases expiration)>0 fails, the server can still find a valid IP address if it can get a response from the clients with return value 0. Hence the strategy of broadcasting an ARP packet to determine if an IP address is available will solve the above-mentioned problem under the assumption that the operating system behaves correctly.

5.5 Verification Effort

The overall verification effort is summarized in Table 5.1. The verification effort was carried out by one graduate student and an undergraduate student working 20 hours a week for about six months. Around 40 percent of this time was spent on the specification while the rest was spent in discharging the refinement proofs. Both the students were initially familiar with Coq; but none of them were experts. Some of the time during specification was spent in understanding the Coq system in greater detail. Significant insight about the router firmware was gained while specifying the system as well as interactively discharging the proofs. This insight was instrumental in nailing down the deviation from the RFC.

Table 5.1: Table for Coq Verification Effort

Implementation C code	2580
RFC Coq specification	≥ 367 lines
C code specification	≥ 620

Chapter 6

Conclusion

In this chapter, we first highlight our contributions of this dissertation. Then, we describe limitations and challenges of this research study, and present open issues for future works.

6.1 Contributions

In this dissertation:

- We developed a novel deductive framework for formal verification of mission-critical embedded software. The framework combines first-order logic theorem proving with automatic SMT solvers.
- We have applied this framework to automatically verify and detect security vulnerabilities in Android applications.
- We also developed a deductive framework for formal verification of MATLAB models. The framework combine static analysis techniques with model-based deductive verification using SMT solvers.
- We provide a formal verification proof of Wireless Router WNR3500L firmware. We report the formal verification of the conformance of the open source Netgear WNR3500L wireless

router firmware implementation to the RFC 2131 based on which it is designed. The formal verification effort led to the discovery of several possible problems in the implementation that we report in this dissertation.

6.2 Limitations and Challenges

In the Java intraprocedural analysis, the constraint system includes all the possible values of variables; this may lead to false positives. More accurate abstract interpretation techniques are required to provide a precise analysis. In the Java interprocedural analysis, we model functions based on summaries; this abstraction loses accuracy and gives out false negatives. Another problem is that our tool needs developers to create specification to specify the security properties of the program; these files may not be easy to construct.

In the wireless router theorem proving verification, we have assumed the correctness of several functions because they are supported by operating systems. However, these assumptions may not be valid in real life; the Coq proof obligations for the program data structures are all based on the assumption that there are no memory operation errors. For a total correctness proof of the systems code, we need to incorporate separation logic-based constructs to model the C program data structure.

In future,we will include the development of a formal system based on this calculus that will enable specification, verification, and automated code generation for more complex software-defined routers such as those based on openflow.

References

[1] http://www.ilovematlab.cn/forum.php.

[2] Andar. http://code.google.com/p/andar/.

[3] Camera class overview. http://developer.android.com/reference/android/hardware/Camera.html.

[4] The coq proof assistant reference manual. http://coq.inria.fr/V8.1pl3/refman/index.html.

[5] Coverity. http://www.coverity.com.

[6] Daisy-epub-reader. http://code.google.com/p/android-daisy-epub-reader/.

[7] Fortify. http://www.fortify.com/.

[8] Forum on risks to the public in computers and related systems. http://catless.ncl.ac.uk/Risks/19.88.html.

[9] Jif:java information flow. http://www.cs.cornell.edu/jif/.

[10] Klock source code analysis for android platform. http://www.klocwork.com/news/press-releases/releases/2008/PR-2008_11_11-Source-code-analysis-for-Android.php.

[11] Nasa mariner 1. http://www5.informatik.tu-muenchen.de/~huckle/bugse.html#mariner.

[12] Netgear wnr3500l. http://www.myopenrouter.com/article/20497/Versiera-Enterprise-Management-and-NETGEAR-WNR3500L.

[13] Npr application. http://code.google.com/p/npr-android-app/.

[14] Opengpstracker. http://code.google.com/p/open-gpstracker/.

[15] Opensudoku. http://code.google.com/p/opensudoku-android/.

[16] Polyspace. http://www.polyspace.com.

[17] Pvs specification and verification system. http://pvs.csl.sri.com.

[18] Smspopup. http://code.google.com/p/android-smspopup/.

[19] Softcheck. http://www.sofcheck.com/products/inspector.html.

[20] Software horror stories. http://www.cs.tau.ac.il/~nachumd/verify/horror.html.

[21] Verified software: Theories, tools, and experiments. http://www.macs.hw.ac.uk/vstte10/.

[22] Steve Hanna Dawn Song Adrienne Porter Felt, Erika Chin and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communication Security*, 2011.

[23] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in Relay Ladder Logic programs. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 184–200. Springer, 1998.

[24] S. Alexander and R. Droms. DHCP Options and BOOTP Vendor Extensions. RFC 2132 (Draft Standard), March 1997. Updated by RFCs 3442, 3942, 4361, 4833, 5494.

[25] Rajeev Alur, Thao Dang, and Franjo Ivancic. Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci.*, 354(2):250–271, 2006.

[26] Paul Anderson, Thomas W. Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, 2003.

[27] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[28] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004)*, pages 1–20. Springer, 2004.

[29] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.

[30] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

[31] Joseph L. Bates. A logic for correct program development. Technical report, Ithaca, NY, USA, 1981.

[32] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[33] Jesse Burns. Developing Secure Mobile Applications for Android: An Introduction to Making Secure Android Applications. Technical report, iSec Partners, October.

[34] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[35] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1st edition, 1997.

[36] Avik Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 1–7, New York, NY, USA, 2009. ACM.

[37] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 160–171, New York, NY, USA, 2006. ACM.

[38] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):pp. 56–68, 1940.

[39] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[40] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[41] Jean-Francois Collard and Ilene Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[42] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[43] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13:451–490, 1991.

[44] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[45] Yuxin Deng and Jean-François Monin. Verifying self-stabilizing population protocols with coq. In *Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '09, pages 201–208, Washington, DC, USA, 2009. IEEE Computer Society.

[46] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006.

[47] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[48] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *ACM SIGSOFT Software Engineering Notes*, volume 19, pages 87–96. ACM, 1994.

[49] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *ACM SIGSOFT Software Engineering Notes*, volume 19, pages 87–96. ACM, 1994.

[50] Jean-Christophe Filliâtre. Formal proof of a program: Find. *Sci. Comput. Program.*, 64(3):332–340, 2007.

[51] Thom Frhwirth and Slim Abdennadher. Principles of constraint systems and constraint solvers, 2005.

[52] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications.

[53] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formaliz. Reason*, 3(2):153–245, 2010.

[54] N. Halbwachs, D. Merchat, and C. Parent-vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *In Static Analysis Symposium, SAS03*, pages 355–365. Springer Verlag, 2003.

[55] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. In *FORMAL METHODS IN SYSTEM DESIGN*, pages 157–185, 1997.

[56] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, 2002.

[57] Markus Hohfeld and Gert Smolka. Definite relations over constraint languages. In *LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80*, 1988.

[58] Gerard J. Holzmann. Software analysis and model checking. In *CAV*, pages 1–16, 2002.

[59] Syed M. S. Islam, Mohammed H. Sqalli, and Sohel Khan. Modeling and formal verification of dhcp using spin. *IJCSA*, 3(2):145–159, 2006.

[60] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[61] Martin Karsten, S. Keshav, Sanjiva Prasad, and Mirza Beg. An axiomatic basis for communication. In *ACM SIGCOMM Conference*, pages 217–228, 2007.

[62] Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23:203–213, April 1997.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[64] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS*, pages 1–12, 2005.

[65] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[66] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 80–92, New York, NY, USA, 1999. ACM.

[67] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. *SIGPLAN Not.*, 45(1):237–248, 2010.

[68] Zohar Manna and Richard Waldinger. *The deductive foundations of computer programming: a one-volume version of the logical basis for computer programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[69] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[70] F. Moller. The specification of an asynchronous router. In *Computer Assurance, 1996. COMPASS '96, 'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 142 –148, jun 1996.

[71] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code.

[72] George C. Necula, Jeremy Condit, Matthew Harren, Scott Mcpeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005.

[73] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[74] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

[75] Corina Pasareanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. In *In Proceedings of TACAS*, pages 284–298. Springer LNCS, 2001.

[76] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Standard), November 1982. Updated by RFCs 5227, 5494.

[77] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for c++ multiple inheritance. *SIGPLAN Not.*, 46:67–80, January 2011.

[78] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. Towards formal analysis of the permission-based security model for android. In *Proceedings of the 2009 Fifth International Conference on Wireless and Mobile Communications*, ICWMC '09, pages 87–92, Washington, DC, USA, 2009. IEEE Computer Society.

[79] David Wagner. Static analysis and software assurance. In *SAS*, page 431, 2001.

[80] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 61–75, Berlin, Heidelberg, 2009. Springer-Verlag.

[81] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety prooffor multiple inheritance in c++. *SIGPLAN Not.*, 41:345–362, October 2006.

[82] Kirsten I. Woldman. A dual programming approach to software testing. Master's thesis, Santa Clara University, 1992.

[83] Weider D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, 3(2):3–21, 1998.

[84] Vincent Zammit. A comparative study of coq and hol. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '97, pages 323–337, London, UK, 1997. Springer-Verlag.

[85] Pamela Zave. Compositional binding in network domains. In *In Proceedings of the Fourteenth International Symposium on Formal Methods*, pages 332–347, 2006.

Vita

Zheng Lu was born in JiangSu ChangShu PR.China, in February 1980. He received the degree of Bachelor in Computer Science from Central South University, China in 2002. He continued his graduate study in Central South University Computer Science Department from 2002 to 2005. Then, he joined the PhD program at Department of Computer Science Utah State University. In 2009, he transferred to Department of Computer Science of Louisiana State University, Baton Rouge. His research interests include formal verification, static program analysis and compile design.