

TABLE OF CONTENTS

Section	Page
1 SUMMARY	1
2 INTRODUCTION	2
2.1 String Subcomponent Taint and Value Analysis	2
2.2 High-level Functional Queries	3
3 METHODS, ASSUMPTIONS, AND PROCEDURES	4
4 RESULTS AND DISCUSSION	5
4.1 String Subcomponent and Value Analysis	5
4.1.1 Points-to analysis	5
4.1.2 String analysis	6
4.1.3 Value analysis	6
4.1.4 Conditional constraints	7
4.1.5 Pruning conditionals	8
4.1.6 Java System Library Model	9
4.1.7 Finite Automaton Analysis	9
4.1.8 Structured Query Language (SQL) Checking	13
4.2 High-level Functionality Queries	15
4.2.1 Overview	15
4.2.2 Tools	15
4.2.3 Examples	28
4.3 String Subcomponent and Value Analysis Evaluation	52
4.3.1 StoneSoup phase 1 tests	52
4.3.2 StoneSoup phase 3 tests	52
4.3.3 Amnesia SQL test suite	61
5 CONCLUSION	65
6 REFERENCES	66
7 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS	68

LIST OF FIGURES

Figure		Page
1	The operation tree built for the code in Subsection 4.1.2. Orange boxes indicate calls that retrieve data from outside of the program. In this case the value of the hypertext transfer protocol (HTTP) get parameters 'username' and 'password'. Blue boxes represent string constants in the program. Green ovals represent string methods.	7
2	The original and exploded operation tree built for the code in Subsection 4.1.8. A separate operation tree is created for each unique combination of set values.	14
3	A snapshot of the current APT transformations.	19
4	A simple example of the tail recursion transformation.	21
5	A simple example of the finite differencing transformation.	21
6	A simple example of the isomorphic data transformation.	22

LIST OF TABLES

Table		Page
1	SQL injection tests from StoneSoup Phase 1. Each test has a single vulnerable SQL statement and other safe SQL statements. Each unique call chain is counted separately. The StoneSoup Phase 1 tests also included five tests that used the Hibernate Database. These were not included as Sansa does not support Hibernate . . .	53
2	SQL injection tests from StoneSoup Phase 3. Each test is generated from a set of features from each category. Each feature is covered in at least one of the tests. . .	60

1.0 SUMMARY

The Sansa project (Static ANalysis for Security Audits) developed static analysis technologies for security audits of Java programs. The system includes two coveted static analysis systems for Java programs: 1) string subcomponent taint and value analysis, and 2) high-level functionality queries. Each is described in more detail below. The combined system provides novel capabilities that substantially increase the precision, accuracy, and throughput of pre-install security auditing of Java-based programs (including Android applications).

String Subcomponent Taint and Value Analysis: We developed an automated static analysis for Java programs that precisely determines possible run-time values and untrusted components of strings. The analysis tracks the source taint (trusted or untrusted) of each subcomponent used to create a string. Untrusted strings are those that are derived from untrusted inputs (such as a network connection)

Furthermore, the analysis reasons about how a string’s untrusted components are sanitized or escaped with respect to a given command language (such as [Structured Query Language \(SQL\)](#)). To achieve additional precision, the analysis accounts for how checks of a string along execution paths constrain the possible values of the string. The analysis is flow and context sensitive which is required in order to analyze common string sanitization and checking code.

We have successfully evaluated the system on a variety of [SQL](#) programs and it was able to find almost all vulnerabilities with limited false positives. Furthermore it was able to provide detailed information provenance for each string subcomponent allowing a user to easily verify and repair vulnerabilities.

High-Level Functionality Queries: We also built a toolkit to help an analyst understand the functional behavior of code segments of interest, by answering questions and extracting facts about the code, including specifications of code behavior. The tool helps in the analysis of tricky segments of code, including those suspected of being incorrect or malicious. It supports reasoning about application-specific behaviors, such as “Can adding a waypoint to a route decrease the total computed time for the route?”. (A malicious route-finding application can add waypoints that take “negative” time, causing the user to take bad routes.) If the answer to such a yes/no question is yes, the tool is often able to produce a concrete input (using the SMT solver) that drives the program to the behavior of interest. The tool also supports scenario-based questions such as “What happens when X is negative?”. In this case, the tool extracts a specification of the behavior that is specialized for the given situation. This makes the extracted specification simpler and more understandable.

2.0 INTRODUCTION

Many commercial, governmental and defense agencies audit third-party software, and they seek to improve the throughput and accuracy of their software security audits. This problem is acute because of at least two demands: (1) supporting bring-your-own-device (BYOD) policies for employees such that personal devices with third-party software can interact with the computing and network environment of agencies, and (2) allowing untrusted third-party developers to contribute to application marketplaces for an agency, enabling more rapid development and capabilities of software. Both of these demands require third-party software to be audited for security threats and vulnerabilities. Our Sansa system can greatly enhance the capabilities of agencies to vet third-party software to determine if it is safe for execution.

The new capabilities will increase throughput, precision, and accuracy of software security audits performed by agencies. Our string analysis will help greatly with a broad range of injection vulnerabilities. Our functionality analysis, and the queries it enables, will greatly enhance the speed and accuracy of understanding complex program functionality (and thus uncovering functionality-based malware).

The proposed research is therefore relevant to organizations that currently audit third-party Java software and more broadly, to any organization that would benefit from more efficient, sophisticated, capable, and cost-effective software.

2.1 String Subcomponent Taint and Value Analysis

Sansa's string component is a whole-program fully context-sensitive and flow-sensitive abstract interpretation. It operates over three value ranges. Object references are supported as set of possible context-sensitive allocation sites (a *points-to* set). Strings are represented as [finite automata \(FAs\)](#). Primitive values are represented as constant values, sets of possible constant values, or unknowns.

Each string operation (e.g., concat, replace, format, etc) is converted into an operation over [FAs](#). Unknown string values (such as those read from an untrusted stream) are represented as a [FA](#) that accepts all strings (i.e., the [FA](#) that corresponds to the regular expression `.*`). Since Java provides an extensive set of string manipulation methods, applications operate at the string level rather than the character level. Java also provides regular expression operations (e.g., `matches`, `replaceAll`, etc). Regular expression operations are easily handled as [FAs](#).

Sansa supports constraints over strings within conditional branches. For example if the check

`str.contains("foo")` is made, the variable `str` will be constrained to contain "foo" in the true branch and constrained not to contain "foo" in the false branch.

Sansa aggressively prunes branches in conditionals and switch statements that are known not be executable. This provides critical precision in general and in particular in the analysis of checking and sanitization code.

[Structured Query Language \(SQL\)](#) checks are implemented by analyzing the list of [FAs](#) that make up a string passed to an [SQL](#) statement. Untrusted inputs within application quotes are checked to ensure that they do not contain any escaped quotes. Untrusted inputs not within application quotes are limited to valid [SQL](#) constants (e.g., true, false, integers, etc). Similar checks could be applied for command injection, cross-site scripting, etc.

2.2 High-level Functional Queries

High level functionality queries about code are enabled by a two step process. First, code is lifted into logic, either by unrolling loops or by turning loops into (tail recursive) logical functions. The result of this step is a representation, in the logic of the [A Computational Logic for Applicative Common Lisp \(ACL2\)](#) theorem prover, of the functionality of the code. Then a variety of tools can be applied to ask questions about the functionality of the code, including yes/no questions, questions that seek an input that causes certain behavior, and scenario-based questions that specialize the functionality with respect to assumptions.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

For all the research we performed in this program, we adopted an experimental approach driven by test cases. Our overarching goal was to produce systems that could successfully process real-world applications and provide useful information for a developer or analyst about those programs.

For the String Subcomponent Taint and Value Analysis, we used test programs developed to test run-time instrumentation approaches to identifying [Structured Query Language \(SQL\)](#) injection. These test programs provided a challenging collection of [SQL](#) applications that use a variety of different input checking and sanitization. The StoneSoup phase 3 tests included a variety of different control flow, data flow, data sources and data types that provided good coverage of different programming approaches.

For the functionality queries, we applied our question answering tools to a variety of test programs. These included small, hand-written programs (e.g., Fermat's Equation, various programs dealing with arrays, and a simplified version of a program that allows HTTP request smuggling). We also applied the tools to various real-world programs, sometimes simplified to remove non-essential features not yet supported by our analysis tools. The real-world examples included a malicious Android app from the DARPA [Automated Program Analysis for Cybersecurity \(APAC\)](#) program and code found in Java projects dealing with Ethereum ([Recursive Length Prefix encoding \(RLP\)](#) encoding) and Bitcoin (Base58 encoding).

We evaluated our technology over different scenarios. We observed any deficiencies and developed techniques that addressed those deficiencies.

The underlying assumption behind this research is that these techniques will generalize to larger classes of programs. We see no way to test these assumptions other than testing our techniques out on a variety of different programs.

During the course of the research, we devoted a major effort to the packaging, testing, and usability of the system. Our releases included both unit and system tests (comprising our regression test). Regression tests were a key component to our development approach; allowing us to easily experiment with different approaches and confirm that the system was still meeting its goals.

All of the systems that we developed run on open source infrastructure (e.g., Linux) and do not require proprietary software to build and run.

4.0 RESULTS AND DISCUSSION

4.1 String Subcomponent and Value Analysis

Sansa is developed on top of the Soot Java Analysis Framework [1]. It processes the Jimple [intermediate representation \(IR\)](#) produced by Soot. Jimple is a typed 3-address [IR](#).

The analysis is a fully context-sensitive and flow-sensitive abstract interpretation. There are three major abstract values supported. Object references are supported as points-to sets. Strings are supported as finite automata. And other primitives are supported as unknown, a constant value or sets of constants. Primitive operations typically result in unknown values.

The analysis is generally sound with the following exceptions:

- Loops are approximated by a single pass. While this can lead to incorrect results, loops are not commonly used in string processing. We encountered one error in the results (see Subsection [4.3.2.6](#)) due to this. More complete support for loops should be a relatively straightforward addition to the analysis
- Exception control flow is not supported. String values of consequence are seldom manipulated in exceptions. We found one example in our test cases where missing exceptional control flow led to a false positive (see Subsection [4.3.3](#))

4.1.1 Points-to analysis

Objects can be allocated directly (via `new` statements) or indirectly via library calls that are not analyzed. Direct allocations yield a precise type while indirect allocations may return a type (such as `Object`) that is not the precise type of the object. For example, `ObjectInputStream.readObject()` returns a value of type `Object` but the actual type is usually much more precise.

Allocation sites (direct or indirect) are fully context sensitive. A separate site is created for each unique context. Each allocation site keeps its own values for its fields allowing very precise values.

Values for references are sets of allocation sites (points-to sets). In the Sansa analysis, points-to sets are immutable allowing assignments to be made efficiently. Operations that would modify a points-to set create a new points-to set with the new values.

4.1.1.1 Casts

When a points-to set is cast, precise types are handled differently from imprecise types. An imprecise type can be cast to any sub-type. Both precise and imprecise types can be cast to a super-type. When a cast is not reasonable (a cast to an unrelated type or a sub-type for a precise allocation type), the allocation site is discarded from the points-to set.

4.1.2 String analysis

Strings are stored as either constant strings or finite automata (when the precise value of the string is not known). Unknown strings (such as those read from files or sockets) are represented as finite automata that accept all strings (i.e., the regular expression `.*`).

The Java types `String`, `StringBuffer`, `StringBuilder`, `CharSequence`, and `Pattern` are treated as strings in the analysis.

When string operations are performed, the analysis does not immediately process the operation. It instead updates an operation tree for the value. Operation trees are evaluated when the code reaches a critical statement (e.g., [Structured Query Language \(SQL\)](#) statements). Thus, string values that are never propagated to a critical statement do not have to be analyzed (building the operation tree is cheap).

For example, consider the following Java code that gets a user name and password from an [hyper-text transfer protocol \(HTTP\)](#) request and builds an [SQL](#) statement with them.

```
String user = req.getParameter ("username");
if (user == "guest")
    passwd = "gpw"
else
    passwd = req.getParameter("password");
sql = "select ... where user='" + user + "'and passwd='" + passwd + "'";
```

The resulting operation tree is in figure 1

4.1.3 Value analysis

Other primitive values (int, float, double, etc) are supported and propagated as constants. When values are joined (e.g., when conditional branches join or when multiple return points in a method are joined) primitive values are stored as a set of possible values. Operations over primitive values result in an unconstrained value.

Primitive values are maintained across assignment and method calls.

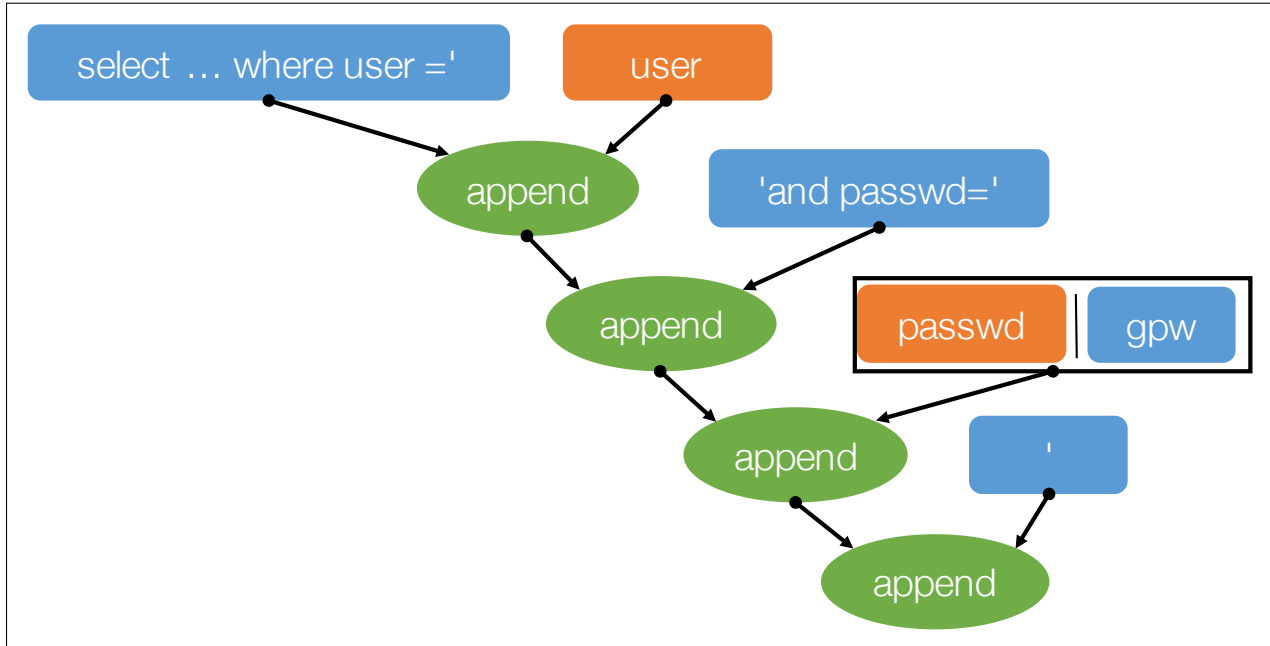


Figure 1: The operation tree built for the code in Subsection 4.1.2. Orange boxes indicate calls that retrieve data from outside of the program. In this case the value of the HTTP get parameters 'username' and 'password'. Blue boxes represent string constants in the program. Green ovals represent string methods.

4.1.4 Conditional constraints

Values are constrained within conditionals. For example, the conditional `str.contains("'')` constrains the value of `str` within the `then` branch to contain a single quote. Conversely, the code within the `else` branch is constrained not to contain a single quote.

These constraints are critical to string processing. Many input checks work in exactly this fashion.

In many cases, the branch is not taken immediately. The result is instead assigned to a boolean (or integer) and the conditional jump occurs in a separate code location. For example, the application may contain a method that checks for valid inputs and returns a boolean indicating whether or not the input is valid.

Sansa supports this by including a `true/false` constraint on boolean values. When those boolean values are later used in a conditional jump, values within the conditional are constrained accordingly. Each value in a conditional branch is compared against the value (or values) that was constrained for the branch's boolean variable. If they match, they are constrained appropriately.

For example, consider:

```
String str1 = str;
boolean b = valid_num(str1);
if (b) {
    // str and str 1 are valid numbers
} else {
```

```

    // str and str 1 are NOT valid numbers
}

boolean valid_num (String s) {
    return s.matches ("[-=]?[0-9]+");
}

```

In the then block both `str` and `str1` are constrained with the regular expression in `valid_num`. Similarly in the else block, `str` and `str1` are constrained to have not matched the regular expression in `valid_num`.

4.1.5 Pruning conditionals

Sansa aggressively prunes conditional branches that it can determine will not be executed in the current context. For example, consider the following code from the Amnesia test case (Subsection 4.3.3)

```

String to\gls{sql}(String value, int type) {
    if (value == null)
        return "Null";

    String param = value;

    switch (type) {
    case adText: {
        param = replace(param, "\\ ", "\\ \\ ");
        param = param.replace("'", "\\ '");
        param = "'" + param + "'";
        break;
    }
    case adNumber: {
        try {
            if (!isNumber(value) || "".equals(param))
                param = "null";
            else
                param = value;
        } catch (NumberFormatException nfe) {
            param = "null";
        }
        break;
    }
    }
    return param;
}

```

The code implements different checks based on the type parameter. Correct processing requires that only the specified case be analyzed. Sansa looks at the values in switch statements and if the value is known, it only analyzes the appropriate branch.

This both improves accuracy and reduces analysis time on branches that are not valid in the current context.

Similar checks are made on `if` statements for null checks, boolean conditions, and integer equality.

4.1.6 Java System Library Model

The system libraries provided by Java are extensive and are widely used in virtually all Java programs. System libraries often have more complex code and utilize more corner constructs than application code. This makes them more difficult to accurately analyze.

Sansa takes two approaches to the Java libraries. For library calls that are not critical to the operation of the application, it applies a very simple model. Calls simply return an unknown value for primitives and strings and a new allocation for references.

Some library calls, however, are critical to the application. For example `java.util` is widely used and application values are stored within the its data types (e.g., lists, sets, etc). These classes are analyzed by Sansa in the same manner as are application classes. As pointed out, however, these classes often contain complex code that both complicates the analysis and lessens precision. To alleviate these problems, a number of the core classes have been simplified. The simpler version removes complexity while still providing an accurate model of the class for the analysis.

For example, `HashMap` was changed from a hash map to a simple array of entries (keys and values). Finding a key is accomplished by enumerating through the array looking for a match. This is much simpler to analyze while providing the same functionality. Obviously, this would run much more slowly, but this is not an issue to a static analysis.

The following classes were modified in `java.util`:

- `AbstractCollection`
- `AbstractList`
- `AbstractMap`
- `AbstractSet`
- `ArrayList`
- `Arrays`
- `Collections`
- `HashMap`
- `HashSet`
- `LinkedHashMap`
- `ResourceBundle`
- `Scanner`
- `TreeMap`
- `TreeSet`

4.1.7 Finite Automaton Analysis

Sansa represents non-constant strings as [finite automata \(FAs\)](#). FAs are able to accurately represent unknown strings and are capable of supporting common string operations in Java programs.

Java provides excellent support for regular expression operations. These are commonly used in Java programs for string manipulations.

FAs are analyzable over Java string operations. Other possible string representations (such as grammars) are not generally solvable unless they are bounded in length.

We use the `dk.brics.automaton` [2] package to implement string operations over FAs. We added support for replace operations over both strings and regular expressions.

The following subsections detail how each of the pertinent string operations are supported. Note that the classes `String`, `StringBuffer`, and `StringBuilder` are all supported as strings. Additionally a subset of `Pattern` and `Matcher` from the `java.util.regex` class are supported as well.

4.1.7.1 Concatenation methods

Concatenation is probably the most common string operation. It is available via the plus (+) operator for `Strings`, the `concat` `String` method and via the various append methods for `StringBuffer` and `StringBuilder`.

Concatenation is a standard FA operation supported by the automaton library.

4.1.7.2 Contains methods

There are a number of string operations that check the value of a string and set a boolean or an integer accordingly. Each of these compares two strings in various ways. In conditionals over the result of these functions (e.g., `if (s1.contains(s2))`), the value of the string is constrained within the conditional. The true and false branches will contain different constraints (depending on the operation). See subsection 4.1.4 for details on how these constraints are implemented.

The contains group of string functions are:

- **contains.** True if string 1 contains string 2. Because `contains` returns true if string 2 is anywhere within string 1, the finite automaton for string 2 is first modified by prepending and postpending a finite automaton that matches any string (i.e., `.*`, resulting in `.*<string-2>.*`). In conditionals where the method returns true, string 1 is constrained to be the intersubsection of the finite automaton for string 1 and the finite automaton of string 2. In false branches, string 1 is constrained not to contain strings 2. This is accomplished by the subtracting the finite automaton for string 2 from that of string 1.

For example, if string 1 is `'[abc]+'` and string 2 is `'a'`, in a true branch, string 1 is constrained to contain at least one `'a'` (i.e., `[abc]*a[abc]*`). In the false branch, string 1 is constrained *not* to contain an `'a'` (i.e., `[bc]+`)

- **equals.** True if the two strings are character by character identical. In a true branch, string 1 is constrained to be the intersubsection of the finite automaton for string 1 and string 2. In a false branch, string 1 is constrained not to be string 2.

- **equalsIgnoreCase**. The same as `equals` except that case is ignored. Each alphabetic character in string 2 is modified to accept both its upper and lower case versions (e.g., 'a' is converted to '[aA]') and then the `equals` analysis is used.
- **compareTo**. Returns -1 if the first string is lexicographically less than the second string, 0 if the strings are equal, and +1 otherwise. For Sansa we treat this the same as `equals` (i.e., we don't differentiate between less-than and greater-than).
- **contentEquals**. The same as `equals` for `StringBuffer` and `StringBuilder`
- **compareToIgnoreCase**. The same as `compareTo` except that case is ignored. Each alphabetic character in string 2 is modified to accept both its upper and lower case versions (e.g., 'a' is converted to '[aA]') and then the `compareTo` analysis is used.
- **matches**. Checks to see if the specified regular expression matches the string. If string 2 is a constant string, that string is treated as a regular expression and converted to a finite automaton. In a true branch, string 1 is constrained to the intersubsection of the finite automaton for string 1 and the automaton created for string 2. In the false branch it is constrained to be the finite automaton for string 1 minus that of string 2.
- **endsWith**. Checks to see if the specified string matches the end of the string. This is the same as `contains`, except that the match-any-string automaton is only prepended to string 2.
- **startsWith**. Checks to see if the specified string matches the beginning of the string. This is the same as `contains`, except that the match-any-string automaton is only postpended to string 2.
- **indexOf**. A non-zero index indicates that the string contains the specified string. This is the same as `contains` except that the returned value is an integer rather than a boolean. The result is the intersubsection for 0 or positive value and minus for a return value of -1.
- **lastIndexOf**. A non-zero index indicates that the string contains the specified string. Same analysis as `indexOf`.
- **isEmpty**. True if the string is empty. Implemented as `equals("")`.

4.1.7.3 Replace methods

There are a number of string operations that modify a string via either implicit or explicit search and replace operations. The searched for values can be either regular expression or strings.

Regular expression replace operations are not a standard finite automaton operation and are not included in the `dk.brics.automaton` library. We added search/replace operations for both strings and regular expression patterns to the library. For regular expressions we used the algorithm used in the PHP analysis tool STRANGER [3]. We wrote an optimized version for constant strings. .

Our replace algorithms do not currently support capturing groups.

- **replace**. Replaces the string (or character) specified in argument 1 with the string (or character) specified in argument 2. This is implemented with our optimized string replace algorithm.
- **replaceAll**. Replace *all* of the occurrences of the regular expression in argument 1 with the string specified in argument 2. This is implemented with our regular expression replace operation.
- **replaceFirst**. Replace the first occurrence of the regular expression in argument 1 with the string specified in argument 2. This algorithm is similar to the `replaceAll` algorithm except that a copy is made of the original finite automaton before the replace operation. All transitions from the ends of the replacement lead to their location in the copy (rather than the original). Since each replacement transitions back to the original finite automaton, the replacement can only occur once.
- **trim**. Remove leading and trailing whitespace. We added a method to the `dk.brics.automaton` library to remove a set of characters from the beginning of a finite automaton. This is essentially the same as `replaceFirst("[\n\r\t]*", "")` with the match constrained to start at the initial state. Trailing characters are removed by reversing the automaton, applying the same transformation and then reversing the result.

4.1.7.4 Misc methods

There are a few miscellaneous methods that didn't fall into any other category.

- **toUpperCase**. Converts all of the characters to upper case. We added this functionality to the `dk.brics.automaton` library. It is implemented by traversing each transition in the finite automaton and converting any transitions on lower case letters to upper case letters. The implementation only supports standard ASCII characters.
- **toLowerCase**. Converts all of the characters to lower case. Same approach as `toUpperCase`.
- **format**. Creates a string by inserting its arguments into the specified format string. If the format string is not a constant, it returns an unconstrained string. If the format string is a constant, it creates a finite automaton for the result based on the arguments passed in the object array. It also uses the type information in the format string to constrain the results. For example, if the format specifier is `'%d'`, the value is constrained to a valid integer (i.e., `'-[0-9]+'`).

Sansa models arrays that are accessed with constant arguments precisely. That allows it to associate the correct value from the object array with the specific format specifiers in the format string.

- **intern**. Returns the current string.
- **split**. Returns an array of strings where each string is constrained to not contain the split regex. This is implemented by a finite automaton minus operation.
- **toString**. Returns the current string.

- **valueOf**. Returns a finite automaton that corresponds to the type of the argument. For example, `getValue(int)` returns the automaton equivalent to `'-?[-\0-9]+'`.

4.1.7.5 Non-string methods

Methods that return non-strings that don't constrain the value of the string (e.g., `contains` returns a non-string, but it constrains the value of its argument) are not modeled. In most cases, they simply return an unconstrained value. This includes methods that return characters or character arrays, these are not currently supported.

- **charAt**. Returns an unconstrained character.
- **codePointAt**. Returns an unconstrained int (codepoint).
- **codePointBefore**. Returns an unconstrained int (codepoint).
- **copyValueOf**. Operates on a character array. Returns an unconstrained string.
- **getBytes**. Returns an unconstrained byte array.
- **getChars**. Returns an unconstrained character array.
- **hashCode**. Returns an unconstrained integer.
- **length**. Returns an unconstrained integer.
- **offsetByCodePoints**. Returns an unconstrained integer (codepoint).
- **toCharArray**. Returns an unconstrained character array.

4.1.7.6 Unsupported methods

A few methods can't be cleanly supported and return an unconstrained value.

- **regionMatches**. Does not constrain its result.
- **substring**. Since there is no clean way to determine an offset within a finite automaton, these return an unconstrained value.

4.1.8 SQL Checking

SQL statements are checked by processing the operation trees for the SQL statement argument to SQL calls (e.g., `java.sql.Statement.execute()`, `java.sql.Connection.prepareStatement()`, etc).

Operation trees are processed in two steps. First each set of values in the SQL statement string is *exploded*. This creates a separate tree for each combination of values in the sets in the string. Consider the following code:


```

user = "guest";
if (login_required)
    user = read_user();

if (user == "guest")
    passwd = "gpw"
else
    passwd = read_passwd();

str = user + passwd;
}

```

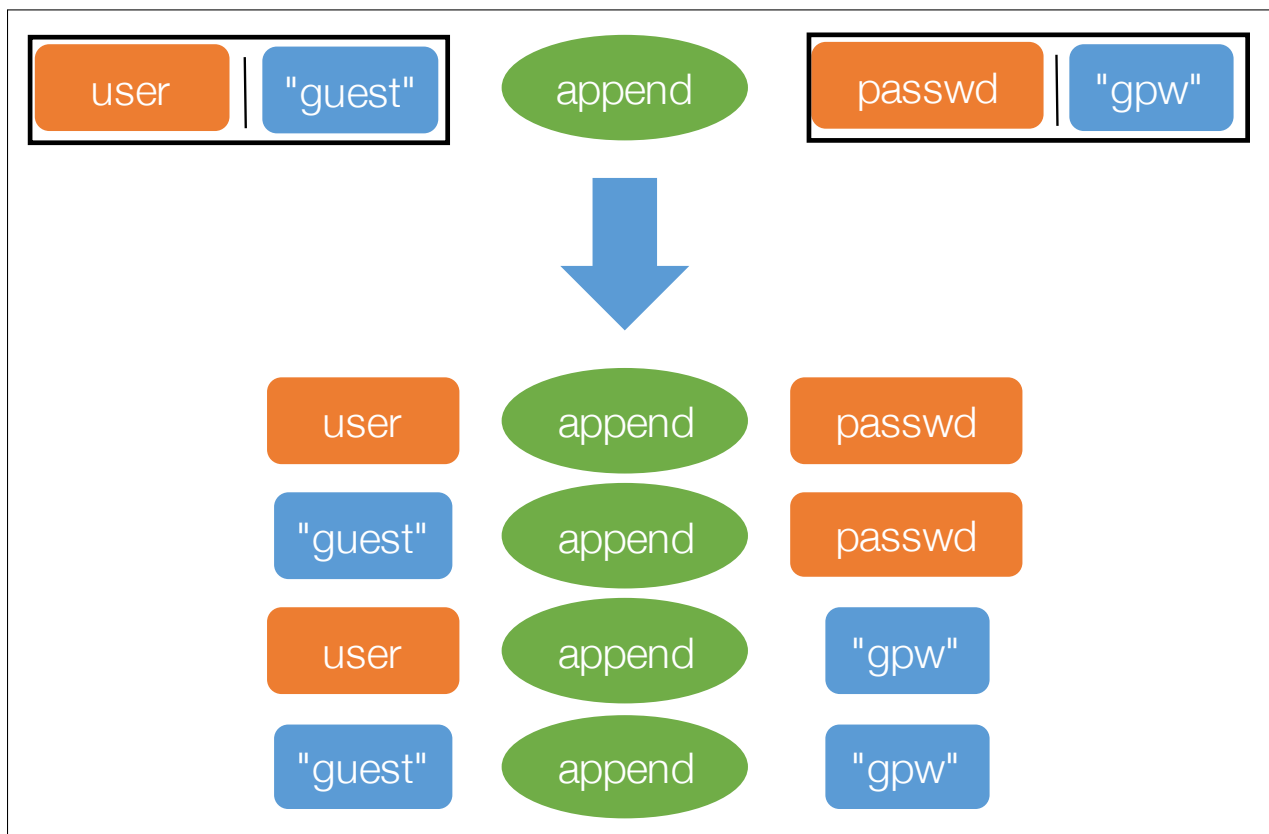


Figure 2: The original and exploded operation tree built for the code in Subsection 4.1.8. A separate operation tree is created for each unique combination of set values.

The variables `user` and `passwd` are both value sets that contain two values each (`user` is either 'guest' or an external (unknown) value, `passwd` is either 'gpw' or an external unknown value). In figure 2 the original tree and the result of exploding the tree are shown. Since there are two values for each variable there are a total of four unique combinations.

Each of the resulting operation trees are now *flattened*. This changes the structure from a tree to a list of regular expressions and/or string constants that make up the string. The flattened list is formed by processing each string method in the operation tree.

Each flattened list is processed from left to right. Each component with external (untrusted) data is checked to ensure that it has been correctly sanitized. Untrusted components that are enclosed in trusted (application) quotes are checked to ensure that they do not contain any embedded quotes that are not escaped. This can be accomplished via a regular expression intersubsection for unescaped quotes. Note that since backslashes themselves can be escaped, the regular expression must check to ensure that an odd number of backslashes precedes any quote:

```
((\.|[^\])*)'|[^\](//)*'
```

Untrusted components that are not enclosed in trusted quotes should be limited to valid [SQL](#) constants (boolean constants or numbers). This can be checked for with the following regular expression.

```
true|false|([-+]?[0-9]*\.[0-9]+|[eE]([-+]?[0-9]+)?).
```

4.2 High-level Functionality Queries

4.2.1 Overview

High level functionality queries about code are enabled by a two step process. First, code is lifted into logic using our [Axe Rewriter](#) and our [Axe Lifter](#), either by unrolling loops or by turning loops into (tail recursive) logical functions. The result of this step is a representation, in the logic of the [ACL2](#) theorem prover[4], of the functionality of the code. Then a variety of tools can be applied to ask questions about the functionality of the code, including yes/no questions, questions that seek an input that causes certain behavior, and scenario-based questions that specialize the functionality with respect to assumptions.

4.2.2 Tools

4.2.2.1 JVM and Android Models

4.2.2.2 Formal JVM Bytecode Model

To reason about a Java or Android program, we compile it to bytecode and reason about how that bytecode executes on a formal model of the [Java Virtual Machine \(JVM\)](#). (If it is an Android app, we intercept its JVM bytecode during compilation.) To assign semantics to this bytecode, we defined in [ACL2](#) a formal model that is an executable interpreter of the Java Virtual Machine [5]. Our model is similar to the M5 model developed by J Moore and others [6], but covers more features (e.g., exceptions, string interning, and class initialization). Theorems about JVM bytecode programs can be expressed using this formal model; we can prove that when the program of interest is executed on the model, starting from a state where certain properties hold, then certain other properties always hold on the resulting state. This follows the style pioneered in [7].

We summarize the JVM model here for concreteness. The state of the JVM in our model includes the Java heap, static area (where static fields are stored), and, for each thread, a call stack that includes invocation frames for each method that the thread is currently executing. Also included are auxiliary data structures for synchronization and locking, string interning, etc.

Each JVM instruction is modeled by specifying the effect on the JVM state when that instruction is executed. For example, the `iadd` instruction for integer addition is modeled as follows:

```
(defun execute-IADD (th s)
  (modify th s
    :pc (+ 1 (pc (top-frame th s)))
    :stack (push (bvplus 32
                  (top (pop (stack (top-frame th s))))
                  (top (stack (top-frame th s))))
                  (pop (pop (stack (top-frame th s))))))))))
```

The function `execute-IADD` modifies the data structures of thread `th` in the JVM state `s`. In particular, it pops two operands off of the operand stack in the top invocation frame of the call stack, adds them, and pushes the sum back onto the operand stack. It then increments the program counter, `:pc`, by 1, which is the length of the `IADD` instruction.

To run an entire program, we repeatedly step the machine state by fetching and dispatching on the next instruction. We use ACL2’s `defpun` utility to soundly introduce the JVM interpreter as a partial function [8].

A crucial feature of our JVM model is that, in addition to running bytecode programs on concrete inputs, it can be used for symbolic execution of bytecode programs on arbitrary inputs. A typical theorem says, in essence, “When we run the JVM model on this bytecode program, for any input satisfying this predicate, the resulting state has the following properties.” The symbolic execution is performed using a rewriter to repeatedly step and simplify the state, symbolically executing one instruction at a time and building up a symbolic representation of the current state in terms of the symbolic inputs. This technique is standard in the ACL2 community. In this way, our formal JVM model captures the semantics of the JVM bytecode language and allows us to reason about Java and Android code.

4.2.2.3 Formal Android Model

We extended the formal JVM model described above to a formal model of the Android platform, capable of executing and reasoning about simple Android apps. A state in the Android model contains a JVM state and several additional Android-specific state components. More precisely, our model of the Android state contains:

- A JVM state, as discussed above. This contains the persistent data used by the app, including its heap and static fields.
- The app’s activity stack, including the current activity on top of the stack, and any activities that are currently paused, below the top activity.

- The set of currently allowed events (e.g., button clicks) for which the app has registered event handlers.
- A parsed representation of the app's manifest.
- The app's layout information, parsed from the app's XML layout files and indexed by the layouts' numeric IDs. This includes information about the views (e.g., buttons) in the app's GUI and their associated event handlers (e.g., `onClick` listeners) and is used by our model of the `setContentView()` [Application Programming Interface \(API\)](#) method when it constructs the GUI for an activity.
- A map from the addresses of `View` objects to their listeners, used to dispatch control when handling events. A listener is a pair of a method (often, but not always, the `onClick()` method of some class) and an object on which to invoke the method (often this is an `Activity` object or an instance of an anonymous class whose sole purpose is to define the listener). This map is updated by our model of the `setOnClickListener()` API method.
- A map from symbolic string names of views, used in the layout XML, to the corresponding numeric resource IDs. This is used to translate events from user-meaningful form to internal form. We build this map by inspecting the static fields of the `R$id` resource class generated when the app is built.
- A map from resource IDs to the addresses of their corresponding `View` objects. This is used to determine the actual objects on which to dispatch events (e.g., click events) and by our model of the `findViewById()` API method.
- The API call history, a ghost variable that lets us talk about the API calls that the app has (and, critically, has not) made.
- The event history, a ghost variable that lets us talk about the sequence of events given to the app so far. If we are verifying that the app implements an abstract state machine, we can abstract this event history and feed it to the abstract state machine. The resulting abstract state should then be the abstraction of the machine's current concrete state. Proving that this property is preserved by all event handlers in the app is a core step of our app proof methodology.
- The event currently being handled, if any, so that we can record in the API history which event was being handled when the API call was made. API calls may be allowed for some events but not others. For example, a sound recorder app may be allowed to start recording only when the user presses the *Record* button.

Event Handling: Our Android model supports running an app on a sequence of input events, by executing their event handlers in order. This can be done on a concrete sequence of events, to test an app. More importantly, it can be used for proof. We prove that, for any sequence of events, running the app's handlers for those events preserves the app's invariant. At this level, events are represented in terms that are meaningful to the user. For example, `(:resume)` represents the event that resumes the current activity, and `(:click"myButton")` represents a click of the button whose name in the layout is `myButton`. In order to actually handle these events, our model must determine the objects on which the handler methods should be invoked, so it first converts the events into an

internal form. For lifecycle events, this adds to the event the address of the topmost activity on the activity stack, giving something like (`:resume12345`). Click events are internalized by mapping the symbolic name of the button to a numeric resource ID and then to the actual address of the View object with that ID, giving something like (`:click6789`). Currently our model only handles lifecycle events and click events, but adding support for other events should be straightforward.

Once the event has been elaborated to internal form, we dispatch it to the appropriate handler by executing the code for the handler using the underlying JVM model. For a lifecycle event, we execute an `invokevirtual` instruction for the appropriate handler method (e.g., `onResume()`) on the given Activity object, which causes the app's `onResume()` handler method to run. Such methods almost always begin by calling through to the corresponding method of the parent class, e.g., `super.onResume()`. This causes code from the Android API implementation to run, e.g., `android.app.Activity.onResume()`. Our model includes special modeling for these lifecycle API calls. For example, the model for `onResume()` causes the `onClick` listeners in the resuming activity to again be added to the set of allowed events. To handle a click event, assuming it is already in internal form, we look up the `onClick` listener for the given View object and call the indicated method. In our model, handlers execute to completion and cannot be interrupted. This corresponds to Android's use of an app's main 'UI thread' to execute its handlers. Future work would include adding support for background services, which an app can use to offload expensive computation from its UI thread.

Events that are not currently allowed by the app (according to the set of allowed events in the Android state) are ignored, e.g., a click on a view that has no registered `onClick` listeners, or an illegal lifecycle event, such as stopping an activity that has not been started. Every event is also recorded in the event history, so that the invariant can refer to the state that the app should be in, given the events seen so far.

API Model: A major challenge in reasoning about Java programs and Android apps is to properly model calls to API methods. We are following a "demand-driven" approach in which we add models of API methods as we encounter calls to them in apps that we want to verify. Some methods such as `sendMessage()` do not really need to be modeled because they affect only the external world, not the state of the app itself: we simply record them in the API history, so that we can express properties such as "the app has not sent any text messages", and continue with execution. When the API call does affect the app's state, if possible we simply execute on our model the actual code of the API from the Android implementation. API calls treated this way include many calls in `java.lang` (e.g., dealing with `Strings` and `Enums`) and setters and getters such as `Activity.setTitle()` and `View.isClickable()`. There are situations where simply executing the API call does not work, either because the code is unavailable (e.g., native methods) or too complicated, or because it affects parts of the Android state that we model. To model such methods, we use executable ACL2 functions that are part of our Android model. Methods that are modeled in this way include `setOnClickListener()`, `findViewById()`, `setContentView()`, and the activity lifecycle event handlers `onStart()`, `onResume()`, etc.

Every run of an app is modeled by building an initial Android state for the app (where many components, such as the API history, are initially empty) and then calling the app's `onCreate()` method.

- tail recursion (several variants)
- undo tail recursion
- finite differencing
- undo finite differencing
- let expansion
- let contraction
- rename parameters
- reorder parameters
- flatten parameters
- drop irrelevant parameters
- drop from mutual recursion
- flip if
- restructure elseif
- make do-while
- lift condition
- weakening
- strengthening
- restore multiple values
- simplify
- simplify if
- predicate narrowing
- wrap input
- wrap output
- homogenize tail recursion
- remove cdring
- data isomorphism
- data expansion (in progress)
- extract output
- extract subfunction
- producer-consumer
- restrict domain
- decrease & conquer
- divide & conquer
- spec substitution
- define spec function
- copy specification

Figure 3: A snapshot of the current APT transformations.

4.2.2.4 APT (Automated Program Transformations)

[Automated Program Transformations \(APT\)](#) [9], is a library of tools, built on the ACL2 theorem prover [4], to transform programs and program specifications with a high degree of automation. APT includes transformations to apply algorithm schemas, turn data into isomorphic representations, apply rewrite rules [10], incrementalize computations, turn recursion into tail recursion, and many others.

APT is useful for program synthesis, to derive provably correct implementations from formal specifications via sequences of refinement steps carried out via transformations. The specifications may be declarative or executable. The APT transformations can synthesize executable implementations from declarative specifications, as well as optimize executable specifications or implementations. The APT transformations can also be used to generate a variety of diverse implementations of the same specification.

APT is also useful for program analysis, to help verify existing programs, suitably embedded in the ACL2 logic, by raising their level of abstraction via transformations that are inverses of the ones used in stepwise program refinement. It is this use of APT that is relevant to the SANSA project, whose goal is to support program understanding, which involves program analysis.

APT enables the user to focus on the creative parts of the program synthesis or analysis process, leaving the more mechanical parts to the automation provided by the tools. APT realizes the classic ideas of program transformation and stepwise program refinement in the state-of-the-art, industrial-strength theorem prover ACL2 (which is used at AMD, Centaur, Oracle, Rockwell Collins, and others).

A snapshot of our currently implemented APT transformations is shown in Figure 3. All of these transformations generate ACL2 proofs of correctness, have been tested, and are being used to develop various correct-by-construction programs, as well as to verify properties of existing programs.

For example, the tail recursion transformation listed in Figure 3 turns certain non-tail-recursive functions into tail-recursive versions.

The variants mentioned in the figure apply to non-tail-recursive functions that satisfy different properties (i.e., applicability conditions of these transformation variants). For instance, the *monoid* variant turns a non-tail-recursive function of the form

$$f(x) = \mathbf{if} \ a(x) \ \mathbf{then} \ b \ \mathbf{else} \ c(d(x), f(e(x)))$$

into a tail-recursive version with an accumulator y

$$f'(x, y) = \mathbf{if} \ a(x) \ \mathbf{then} \ y \ \mathbf{else} \ f'(e(x), c(d(x), y))$$

provided that the operator c that combines the result of the recursive call with (a function of) the argument is associative and has b as left and right identity (i.e. c and b form a monoid, hence the name of the transformation variant). The transformation also generates a theorem of the form

$$f(x) = f'(x, b)$$

that asserts the correctness of f' with respect to f .

Tail recursion in logical and functional languages corresponds to iteration (i.e. loops) in imperative languages. Thus, the tail recursion transformation is important to formally bridge the gap between specifications that often use non-tail-recursive functions (which are often clearer and easier to prove properties of, compared to their tail-recursive counterparts) and imperative code to be synthesized or analyzed.

Figure 4 shows a very simple example of use of the tail recursion transformation. The transformation is applied to definition of the factorial function that mirrors the typical mathematical definition, recursive but not tail-recursive. The result is an equivalent tail-recursive version that uses an accumulator for the result, as typical in functional programming. The new function and the theorems that prove it equivalent to the old function are automatically generated by the tail recursion transformation.

Note that, in the list in Figure 3, the tail recursion transformation is immediately followed by ‘undo tail recursion’. This is the “inverse” of tail recursion transformation: it turns a tail-recursive function (like f' above) into a non-tail-recursive equivalent (like f above), thus raising the level of abstraction and easing formal reasoning. In the example in Figure 4, the ‘undo tail recursion’ transformation goes “up”, from the tail-recursive version of factorial to the higher-level, mathematical, non-tail-recursive formulation.

Figure 5 shows a very simple example of use of the finite differencing transformation (which is listed in Figure 3). The finite differencing transformation caches partial results to speed up repeated computations. In the figure, this transformation is applied to a function that recursively computes sum of squares. The transformation automatically generates a version of the function that caches the square of the argument n and avoids multiplications (which could be useful on a platform where multiplication is more expensive than addition and subtraction; this is just a simple demonstrative example).

Tail Recursion Transformation: Factorial Example

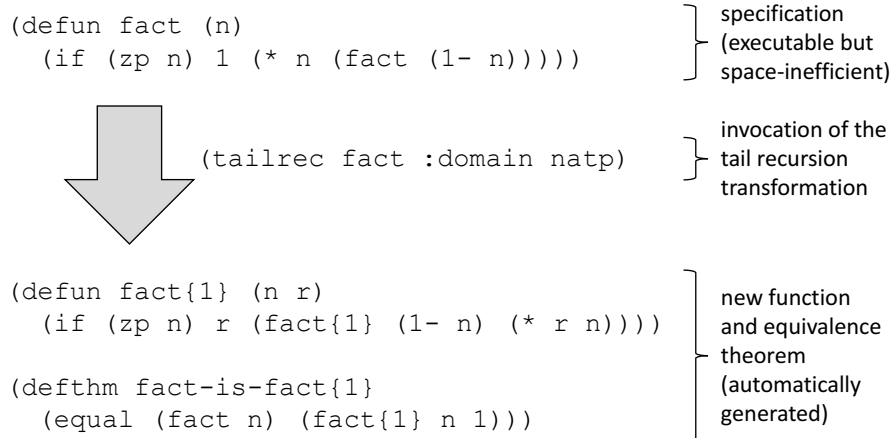


Figure 4: A simple example of the tail recursion transformation.

Finite Differencing Transformation: Sum-of-Squares Example

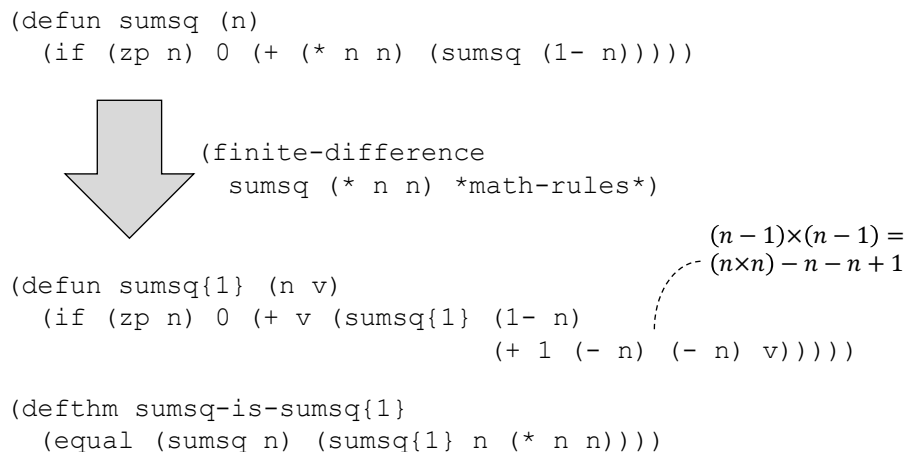
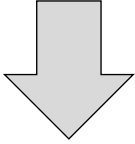


Figure 5: A simple example of the finite differencing transformation.

Isomorphic Data Transformation: Integers-to-Naturals Example

```
(defun double (x) (* 2 x))
```



```
(isodataf double x  
  integerp natp  
  int-to-nat nat-to-int)
```

```
(defun double{1} (x) (* 2 (nat-to-int x)))
```

```
(defthm double-is-double{1}  
  (implies (integerp x)  
    (equal (double x)  
      (double{1} (int-to-nat x)))))
```

Figure 6: A simple example of the isomorphic data transformation.

Programs often cache information for efficiency, at the cost of complicating formal reasoning. The ‘undo finite differencing’ transformation in Figure 3, listed just after the finite differencing transformation, can “undo” some of these caching optimizations, producing functions that are easier to formally reason about. This is the “inverse” of the finite differencing transformation: in the example in Figure 5, the ‘undo finite differencing’ transformation goes “up”, from the two-argument version of *sum-of-squares* (whose second argument contains cached information) to the simpler, one-argument version.

Figure 6 shows a very simple example of use of the isomorphic data transformation (which is listed in Figure 3). The isomorphic data transformation changes the representation of data (e.g., the arguments of a function) into isomorphic representations. The figure shows an artificial example where integer numbers are changed into natural numbers (according to an isomorphic mapping between the two sets of numbers), but this transformation is very useful in real-world cases, e.g., to perform a large class of data type refinements.

Since isomorphisms are, by definition, invertible, the isomorphic data transformation is its own “inverse”, i.e. there is no separate ‘undo data isomorphism’. Rather, the same transformation can be run with the arguments for the old and new representations and for the conversions swapped. For example, in Figure 6, swapping *integerp* with *natp* and swapping *int-to-nat* with *nat-to-int* in the call to *isodataf* goes “up”, changing the data representation from natural numbers to integer numbers. Again, the example in Figure 6 is artificial, but a more realistic use of this transformation is to change the representation of data from the Java integer types to the theorem prover’s higher-

level arithmetic types. This is generally carried out next to another transformation (‘simplify’ in Figure 3) that replaces, via rewriting techniques, modular arithmetic operations (as defined in Java) with the exact arithmetic operations of the theorem prover. The latter are easier to formally reason about than the Java operations.

4.2.2.5 The Axe Rewriter

Axe contains a sophisticated rewriter capable of efficiently transforming large terms by repeatedly applying local “rewrite rules.” The rewrite rules used are equivalence-preserving. That is, they allow a term to be replaced only by another term which always returns the same value. Because Axe terms are side-effect-free, rewriting can be performed without changing the meaning of any over-arching term.

A major goal of rewriting is to simplify the terms involved. Simplification is almost always desirable; it helps keep terms small and is sufficient to completely prove some claims (because they simplify down to “true”). A related goal is to normalize terms. Normalization is the process of transforming syntactically different but semantically equivalent terms so that they have the same syntactic representation (and so are clearly equal by inspection). Efficient rewriting schemes may not exist to completely normalize the terms with which Axe deals. Nevertheless, normalization works fairly well in practice, using the techniques and the rewrite rules described here.

Rewriting can also be used to simply change the form of terms, for example, to bit-blast them (to split multi-bit operations into single-bit operations), or to eliminate certain operators (such as the `leftrotate` operator, which cannot always be directly translated into the language of the [Simple Theorem Prover \(STP\)](#) solver). Finally, rewriting can be used to strengthen a set of known facts, using information provided by some new fact.

The Axe Rewriter is similar to the rewriter of the ACL2 theorem prover [4] and borrows many important ideas from it. However, the Axe Rewriter has several advantages. The most significant is its efficient representation of terms as directed acyclic graphs (DAGs). In the DAG representation, each distinct subterm is represented only once. Nodes in the DAG represent expressions and are numbered sequentially from 0. The allowed expression types are constants, variables, and functions applied to lists of arguments, each of which is a constant or the number of another node. The DAG is acyclic because a function call node’s arguments must be nodes with smaller numbers.

The Axe Rewriter also performs memoization, uses “objectives” to guide rewriting, efficiently handles large XOR nests, provides more fine-grained control of rule ordering, and provides a facility for calling the Axe Prover (a separate but related tool) to discharge hypotheses of conditional rewrite rules.

The Axe Rewriter is very fast, performing, in one example, about 600,000 rewrite rule attempts per second.

The Axe Rewriter is designed to be used as a stand-alone component, in contrast to the ACL2 rewriter, which is interwoven with the rest of ACL2’s proof process. The Axe Rewriter is designed to be trusted but is not proved correct with ACL2 (although that could be attempted some day).

An alternative to using a rewriter in a verification system is to simply encode all the desired transformations by hand; one would simply write a program that transforms terms (DAGs) in whatever way is desired. However, this is likely to be complicated and error prone. Adding a new transformation would require writing more code, and programming bugs could compromise the correctness of the system. Axe’s approach is to factor the problem into two parts, a general-purpose rewriter and a set of rewrite rules (theorems) used to “program” the rewriter. This approach has several advantages. First, it allows a large number of different transformations to be added to the system; Axe contains hundreds of rewrite rules, and it would be tedious to write code for all of them. Second, Axe’s approach makes it easy to experiment with different sets of transformations; one simply passes different sets of rewrite rules to the Axe Rewriter. Similarly, one can easily change the order in which transformations are applied, using Axe’s system of rule “priorities”. Finally, adding new transformations can be done in a sound way because the transformations can be proved as theorems in the ACL2 logic. This ensures that adding a new rule won’t render the system unsound. When new rules are added, the amount of the code that must be trusted (the code of the Axe Rewriter itself) stays the same.

The Axe Rewriter lacks several features of the ACL2 rewriter, including “forward-chaining” rules, a procedure to determine the types of terms, and a built-in linear arithmetic procedure. None of these features were necessary for the examples discussed here.

4.2.2.6 Calling the STP SMT Solver

We have developed a connection between the ACL2 theorem prover, on which most of our tools are based, and the [STP Satisfiability Modulo Theories \(SMT\)](#) solver[11]. This allows us to take advantage of STP’s highly efficient reasoning about the theory of bit-vectors and arrays (which can model many commonly-used programming constructs). The process begins by applying the Axe rewriter to simplify the DAG to be proved as much as possible with respect to assumptions provided by the user. It then represents the full claim to be proved as a disjunction, attempting to prove that either the conclusion is true, or one of the assumptions is false. The tool must address the fact that the DAG may not entirely consist of operations in STP’s theory. (STP handles operators over bit-vectors, such as BVPLUS, and bit-vector arrays, such as BV-ARRAY-READ. Other operators must be abstracted away to variables of types that STP can handle (possibly also booleans). The abstracted nodes become variables in the STP query, so the tool also must determine types (that STP can handle) for them. It does this using two sources of information: information implied by negated disjuncts (which is safe to use since if any disjunct is actually true, then it suffices to prove the entire disjunction), and information determined by looking at how the terms to be abstracted are used (e.g., how many bits of the term are ever used, which is safe because the bit-vector operators carry explicit size parameters and “chop” their arguments down to size before operating on them). The translation to STP also supports heuristically cutting out (abstracting) nodes in the DAG deemed unlikely to contribute to the proof; it can even perform a binary search on the “cut depth” to attempt to find an appropriate abstraction level (abstract away too much and the STP goal may no longer be true, abstract away too little and the solver may time out).

In this project, we improved the connection to STP in several ways, including adding the ability to parse, process, and return the counter-examples found by STP. This forms the basis of our

query answering capability; we pose a query to STP that attempts to prove that some behavior is impossible, and it returns a concrete input showing when the behavior is in fact possible.

4.2.2.7 The Axe Lifter

The Axe Lifter lifts the functionality of Java code into a logical representation. It does so by applying the Axe Rewriter to perform symbolic simulation of the JVM model as it executes the program on symbolic input. The result is a mathematical term that represents the effect of the code, expressing the code's output in terms of its symbolic inputs. For compactness, term is represented as a structure-shared DAG (directed acyclic graph) in which each node is a constant, variable, or a named function applied to other nodes.

There are two main approaches to lifting code, depending on how loops are handled. In some cases, loops can be fully unrolled, because their iteration counts are fixed or bounded by a known bound (either in all cases, or because of assumptions given to the lifter). The result of lifting is then a loop-free representation of the code's functionality in logic. This is implemented in the tool `unroll-java-code`. When loops cannot be unrolled, they can often be lifted into tail recursive functions. This style of lifting is implemented in the tool `lift-java-code`, which we call the non-unrolling lifter. Note that this style of lifting is less automatic, because typically loop invariants are needed, e.g., to show that exceptions do not occur in the loop body or that different storage locations manipulated by the loop do not alias. The lifter can automatically guess and check simple invariants.

Normally, `unroll-java-code` and `lift-java-code` effectively inline any subroutines encountered. It may be desirable to instead lift code compositionally (one subroutine at a time). This is implemented for the unrolling lifter as the tool `unroll-java-code2`. (It is not yet implemented for the non-unrolling lifter but may be straightforward.) A related tool, `lift-java-code-segment` can lift a fragment of Java code that does not form a complete method.

Some restrictions apply to lifting. First, recursion is only supported in the unrolling lifter (though we may add support for it in the non-unrolling lifter as well). Second, the non-unrolling lifter can only lift loops that touch a finite set of heap objects. This is because each field touched by the loop becomes a parameter of the function that represents the loop. Furthermore, the lifter must be able to show that the object fields touched by the loop cannot alias. This is eased somewhat by the fact that fields whose names or types differ cannot possibly alias each other.

The lifters allow the user to indicate which part of the JVM state holds the output of interest after execution. Typically, this is the return value of the function or the final value of some field (e.g., the contents of one of the array objects passed into the function). This becomes the return value of the function produced by the lifter.

The lifters, perhaps with guidance from the user, determine which JVM state components affect the output of the program to be lifted. These become parameters of the function produced by the lifter.

The lifters allow assumptions about the inputs to be supplied and allow rewrite rules to be passed in for simplifying the terms that arise during lifting. The loop lifter allows the user to supply

information about the loops to be lifted, including invariants, type information, and termination measures. The user can also supply arbitrary properties (in the form of ACL2 theorems called loop postludes) to be proven after one loop before lifting the next loop; these are needed because values from one loop may flow into a subsequent loop, and the lifter may need to show that such values satisfy the invariant of the second loop.

The loop lifter requires a loop body to provably never throw an exception or error (assuming that the loop invariant holds), but settings are available to discard execution paths that result in exceptions or errors. Doing so is unsound but sometimes helpful in quickly lifting a loop.

The loop lifter attempts to lift a loop using a set of candidate invariants (some generated by tool and some supplied by the user). This includes lifting any nested loops. It then tries to show that the invariants assumed are in fact preserved by the loop body (assuming the loop does not exit). If this proof fails, the lifter discards the invariants that failed to prove and attempts again to lift the loop. The process continues until an inductive set of loop invariants is found.

The function that represents a lifted loop contains an exit test (representing the conditions under which the loop terminates) and an update function (representing the change made to the loop variables by the loop body). The loop function repeatedly executes the update function until the exit test is true, whereupon it returns the loop parameters as a tuple.

After lifting, the term (actually a DAG) representing the lifted program can be subjected to further analysis (e.g., theorem proving, equivalence checking, or test generation). Often the result of the unrolling lifter contains only simple operators over bit-vectors and arrays, allowing highly automated reasoning using our connection to the STP SMT solver. The output of the non-unrolling lifter contains recursive functions that model the loops; these can be transformed using our APT transformation toolkit and subjected to analysis by theorem proving. When a loop contains no nested loops its body is loop-free and can thus be subjected to highly automated SMT-based reasoning.

4.2.2.8 Query tool: ‘Prove-with-tactics’

Prove-with-tactics (also called the tactic prover) takes a term and attempts to prove it true (i.e., non-nil) for all assignments of values to its variables. Assumptions (terms to assume true during this process) can be supplied. The tactic prover traffics in proof problems. A proof problem is just a list containing the term to prove (in DAG form) and a list of assumptions.

The tactic prover operates by applying a sequence of proof tactics to the term, in order. Each tactic takes a proof problem and returns a “tactic result”, which is one of the following:

- `:valid`, meaning that the tactic proved the goal
- `:invalid`, meaning that the tactic determined that the goal is not provable
- `:nochange`, meaning that the tactic was unable to make progress
- `:error`, meaning that an error occurred

- a list whose first element is `:problems` and whose other elements are proof problems, indicating that the tactic split the problem into subproblems such that if all the subproblems are provable then the original problem is provable

A tactic can also return additional information, such as a counterexample for an invalid proof problem.

Tactics are tried in order until a tactic returns `:valid`, `:invalid`, or `:error`, or until no more tactics remain. When a tactic returns multiple sub-problems, the entire list of remaining tactics is applied to each of them. Note that a tactic commonly returns a single sub-problem that is simpler than the original (because simplification or pruning has taken place).

Supported tactics include:

- `:rewrite` – call the Axe rewriter
- `:prune` – drop unreachable branches, using full contextual information and (optionally) calling STP on if-tests to try to prove that branches are infeasible
- `:prune-with-rules` – same as `:prune` except the Axe rewriter is applied (with the given rules) to attempt to rewrite each test to true or false
- `:cases` – split into cases (sub-problems), after showing that the given cases are exhaustive
- `:stp` – attempt to use STP to prove the goal
- `:acl2` – attempt to prove the entire goal with ACL2

When testing the tactic prover, we use two utilities, `must-fail` and `must-succeed` to indicate the expected results of attempted proofs. Wrapping a form, such as an attempted proof with the tactic prover, in `must-fail` ensures that the proof attempt does indeed fail (and continues to fail even as we evolve our tools). Likewise, `must-succeed` ensures that the proof attempt succeeds, and continues to do so as we change and re-test our tools.

4.2.2.9 Query tool: ‘Query’

The QUERY tool takes a term and tries to find an assignment of values to its variables that makes it true, or tries to prove that no such assignment exists. QUERY operates by rewriting the given term and then calling the STP SMT solver. The term should usually contain only operators supported by STP (which handles bit-vectors, booleans, and fixed-size arrays of bit-vectors). Subterms that are not translatable to the language of STP will be abstracted to variables of those types. Thus, if nothing in the query is expressible in STP, the entire term will be abstracted away! Due to abstraction, assignments reported by QUERY may be spurious, but the tool prints details of any abstraction done. If QUERY proves that the abstracted goal is unsatisfiable, the original (less general) term without abstraction should also be unsatisfiable.

QUERY works by calling the tactic-based prover to try to prove the negation of the term given by the user. It always uses the tactic `:rewrite` followed by the tactic `:stp`. If the tactic prover proves the negation of the term, then there is no assignment that makes the original term true. If the tactic

prover fails to prove the negation of the term, and returns a (non-spurious) counterexample, then that counterexample is an assignment that makes the original term true.

The `:rules` option can be used to supply names of rewrite/simplification rules (theorems to apply and functions to expand) for `Axe` to apply before calling `STP`. This can help `Axe` transform the term to have a form that is more amenable to translation to `STP` (e.g., by re-phrasing things in terms of the operators `STP` can handle). Rewriting may also be used to prove or disprove the term by rewriting it to a constant (true or false).

Examples:

The query `(query (equal (bvplus 8 1 x) 3))` asks whether there is an (8-bit) value x such that the (8-bit) sum of x and 1 is 3. `QUERY` finds the assignment $x=2$.

The query `(query (equal (bvplus 8 1 x) (bvplus 8 2 x)))` asks whether there is an (8-bit) value x such that the (8-bit) sum of x and 1 is equal to the (8-bit) sum of x and 2. `QUERY` proves that there is no such x .

4.2.3 Examples

This section demonstrates the application of our toolkit to a variety of examples. First, some artificial examples demonstrate the basic capabilities. Then we discuss the HTTP request smuggling and malicious route finding examples, where queries reveal security issues and malware, respectively. Finally, we discuss RLP and Base58 encoding, as further demonstrations that the tools can analyze code in a deep, application-specific way and can find bugs and other interesting behaviors. The Base58 examples also show the use of our tools to perform specialization, to explore the behavior of code in specific scenarios.

4.2.3.1 Example: Fermat's Equation

We applied our tools to find counterexamples to Fermat's Last Theorem when arithmetic is performed modulo 2^{32} . This is an artificial example meant to be very simple yet demonstrate interesting capabilities of the tool. Fermat's Last Theorem (dating back to 1637) asserts that there are no positive integers x , y , and z such that $x^n + y^n = z^n$ for $n > 2$. But of course, the arithmetic used (the addition and exponentiation) is assumed to be infinitely precise, with no upper bound on the numbers involved. In contrast, programming languages such as Java typically natively support fixed-width arithmetic (e.g., Java's 32-bit ints).

The following Java method tests whether x , y , and z represent a counterexample to Fermat's conjecture for $n = 3$ when the arithmetic is performed modulo 2^{32} :

```
// Fermat's equation when n = 3. Test whether x^3 + y^3 = z^3.
static boolean fermatEquation (int x, int y, int z) {
    return x * x * x + y * y * y == z * z * z;
}
```

We can apply our query tool to find inputs that cause the method to return true. First we lift the code into logic:

```
(unroll-java-code *fermat* "Fermat.fermatEquation(III)Z"
                  :extra-rules '(run-until-return-from-stack-height-of-myif-split))
```

This yields a logical function $fermat(x,y,z)$ that represents the Java code, testing whether $x^3 + y^3 = z^3$. We can query it to find values of its inputs that cause it to return true. For example, the query

```
(query (equal 1 (fermat x y z)) :rules '(fermat))
```

finds the counterexample:

```
x=0
y=0
z=0
```

We need to exclude 0 since Fermat's theorem deals with positive numbers. So we add assumptions to the query to exclude such cases:

```
(query
  (and (sbvlt 32 0 x) ;; signed bit-vector less-than, means 0<x
        (sbvlt 32 0 y)
        (sbvlt 32 0 z)
        (equal 1 (fermat x y z)))
  :rules ...)
```

This finds the counterexample (using hexadecimal notation):

```
x=0x10000000
y=0x00000001
z=0x00000001
```

Note that overflow occurs when cubing 0x10000000, yielding 0. So this amounts to $0 + 1 = 1$, which is still not very interesting. We add another assumption excluding $x^3 = 0$:

```
(query
  (and (sbvlt 32 0 x)
        (sbvlt 32 0 y)
        (sbvlt 32 0 z)
        (not (equal 0 (bvmult 32 x (bvmult 32 x x))))
        (equal 1 (fermat x y z)))
  :rules ...)
```

This gives a more interesting counterexample:

```
x=1819708793 (decimal)
y=6
z=1
```

If we check it, we find that the cube of 1819708793 is 6025674680791187200222953257, of which the low 32 bits (Java chops the result after multiplying) are 0xFFFFFFFF29. This represents the value

-215 in signed twos-complement format. When -215 is added to 6^3 (i.e., 216) we indeed get 1^3 , which is 1. So the tool found an interesting counterexample which was not obvious to the human.

As a final example, we can disallow small values of the variables (such as the 1 and 6 found above):

```
(query
  (and (sbvlt 32 1000000 x)
        (sbvlt 32 1000000 y)
        (sbvlt 32 1000000 z)
        (not (equal 0 (bvmult 32 x (bvmult 32 x x))))
        (equal 1 (fermat x y z)))
  :rules ...)
```

Here the tool finds something even more interesting:

```
x=177496761 (decimal)
y=1000006
z=1000001
```

Even this last query takes under a second to run. These simple experiments show the power of the SMT-backed query tool to automatically find interesting values that satisfy the logical formulas extracted from programs.

4.2.3.2 Example: HTTP Request Smuggling

We applied our tool to a program that exhibits an issue similar to the HTTP request smuggling vulnerability. HTTP request smuggling occurs when it is possible to craft an HTTP message that is interpreted differently by a server than by a firewall protecting that server. Typically, this is done by including more than one Content-Length header in the message. The problem occurs if the server and firewall differ on which of the multiple Content-Length headers is actually used. In particular, the firewall may see one message while the server sees two messages. In this case, the second message will be invisible to the firewall (it will consider those bytes to simply be the body of the single message that it sees). Thus, the second message can reach the server without being checked.

Consider the Java program:

```
static byte[] parseWithOverride (byte[] msg) {
    byte[] values = new byte[256];
    int i = 0;
    while (i + 1 < msg.length) {
        byte key = msg[i];
        byte value = msg[i+1];
        values[key] = value;
        i += 2;
    }
    return values;
}
```

```
}
```

This program parses an input message which contains alternating one-byte keys and one-byte values into a 256-byte value array indexed by keys. We lift this program into logic, assuming that the input, *msg*, is 10 bytes long:

```
(unroll-java-code *parseWithOverride* "Test.parseWithOverride([B][B]"
  :extra-rules
  '(run-until-return-from-stack-height-of-myif-split)
  :vars-for-array-elements nil
  :array-length-alist '((msg . 10))
  :ignore-exceptions t)
```

Now we would like to prove that if the message contains a particular *key* (say, at byte 6 in the message), immediately followed by a particular *value* (say, at byte 7), then that *value* is indeed the *key*th byte of the result. This is represented by a query like the following:

```
(prove-with-tactics
  '(implies (and (bv-array-p 8 10 msg)
                 (equal key (bv-array-read 8 10 6 msg))
                 (equal value (bv-array-read 8 10 7 msg)))
            (equal (bv-array-read 8 256 key , (dag-to-term *parseWithOverride*))
                  value))
  :tactics '(:rewrite :stp))
```

This query fails! In fact, the tool finds the counterexample:

```
(Counterexample:
 Node 0: MSG is (0 0 0 0 0 0 0 1 0 0).
 Node 23: (UNSIGNED-BYTE-P-LIST '8 MSG) is T.
 Node 25: (TRUE-LISTP MSG) is T.
 Node 27: (LEN MSG) is 10.
 Node 30: KEY is 0.
 Node 33: VALUE is 1.)
```

Note that when the input message contains duplicate keys, later values will override previous values for the same keys. The counterexample message contains a key value pair of (0, 1) starting at position 6. This causes the value 1 to be stored at position 0 in the result. But message contains a later pair of (0, 0) starting at position 8. This causes the value 0 to override the 1 previously stored at position 0. This example input found by the tool illustrates the essence of the HTTP request smuggling problem: A message encodes key value pairs, and duplicate keys cause later values to overwrite earlier values for the same key. It is then critical which of the two values implementations choose to use, and if they differ on this point, problems may ensue.

We applied our query tools to various variations of this problem, including a version in which later values are not overridden (where it is dangerous if implementations do not respect that convention) and variants where the message formats were more complicated. Our SMT-based tools help clarify such code by producing counterexamples that falsify desired properties, or by showing that no such examples exist.

4.2.3.3 Example: Route Finding

We applied our query answering tools to an example of a malicious route finding application. This example, called “koenigsberg”, was a simplified version of an app provided by the Red Team on the DARPA APAC program. That app was a major motivating example for this project. The malware involves calculating very bad routes when certain types of “places” are available during route planning. Essentially, certain places are considered by the route finder to take negative time to visit (!), causing routes that visit them repeatedly to appear very cheap.

The relevant code includes this Java method:

```
int addStopTime(Place stop, int stopTimes) {
    int extratime = 0;
    PlaceType[] stoptypes = stop.getTypes();
    if (stoptypes != null) {
        for (PlaceType s: stoptypes) {
            int weight = s.getWeight();
            extratime = extratime + weight;
            extratime = extratime * Integer.signum(weight);
        }
    }
    extratime = extratime * Integer.signum(stopTimes);
    return stopTimes + extratime;
}
```

which our analyst found confusing. What is this code doing? Apparently a stop can have multiple types, and the time spent at the stop is some function of all of its types. We might expect it to just be the sum, but the code is doing something more complicated.

Our analyst conjectured that it might be possible for this method to return a result smaller than the *stopTimes* passed in, but it wasn’t clear whether or exactly how (i.e., for what values of *stop.getTypes()*) this could happen. The query tools that we developed on this project allow such questions to be answered.

We lifted the code into logic, assuming there are 5 stop types. This gives us an expression over the variables *stoptimes*, *type0*, *type1*, *type2*, *type3*, and *type4*. We can then pose queries involving those variables and the result computed by *addStopTime*. For example, this query:

```
(prove-with-tactics '(not (sbvlt 32 ,(dag-to-term *add-stop-time*) stoptimes))
:tactics '(:rewrite :stp))
```

attempts to prove that *addStopTime* never decreases the provided *stoptimes*. The proof fails with the counterexample:

```
(Counterexample:
Node 0: STOPTIMES is 2147483647.
Node 1: TYPE0 is 0.
Node 9: TYPE1 is 0.
```

```
Node 18: TYPE2 is 0.
Node 27: TYPE3 is 0.
Node 36: TYPE4 is 0.)
```

This counterexample is perhaps not very interesting because it requires *stoptimes* to already be the maximum representable integer. We can exclude that case as follows (where we assume *stoptimes* is less than one million):

```
(prove-with-tactics '(implies (sbvlt 32 stoptimes 1000000)
                             (not (sbvlt 32 ,(dag-to-term *add-stop-time*) stoptimes))
                             :tactics '(:rewrite :stp)))
```

Again the tool finds a counterexample (omitted here), but this time it requires *stoptimes* to be initially negative, which may not be possible. So we exclude that case too, giving the query:

```
(prove-with-tactics '(implies (and (sbvlt 32 0 stoptimes)
                                   (sbvlt 32 stoptimes 1000000))
                             (not (sbvlt 32 ,(dag-to-term *add-stop-time*) stoptimes))
                             :tactics '(:rewrite :stp)
                             :print :verbose)
```

Now the tool finds an interesting counterexample:

```
(Counterexample:
Node 0: STOPTIMES is 2.
Node 1: TYPE0 is 5.
Node 9: TYPE1 is 37.
Node 18: TYPE2 is 0.
Node 27: TYPE3 is 0.
Node 36: TYPE4 is 0.)
```

It turns out that the weight of a stop (defined in a separate file) depends on looking up a value in this array:

```
static final int weights[] = {
-1,
600,
1200,
7200,
7200,
1200,
120,
300,
...
1200,
900,
300,
3600,
300,
```

```
300,  
1200,  
5400,  
1800,  
7200,  
5  
};
```

Note that element 0 of this array is -1 . Since the types of stops 2, 3, and 4 in the counterexample are 0, this -1 is used in the complicated computation in `addStopTime`, ultimately causing it to decrease the total time of the route. So the tool helps the analyst by seeing through the complexity of the calculation and discovering an input that shows that, yes, the suspected bad behavior is indeed possible.

The extra complexity of `addStopTime` (e.g., the calls to `signum`) seems to have been inserted just to confuse the security analyst. The example input provided by the tool spares him or her from having to understand such convoluted code.

We performed a final query that disallowed 0 as a stop type (and constrained the stop types not to be too large). With those assumptions, the tool can prove that `addStopTime` cannot decrease its argument. So the helps show that the -1 at element 0 in the array really is essential to the bad behavior.

4.2.3.4 Example: Recursive Length Prefix encoding

[Recursive Length Prefix encoding \(RLP\)](#) is an encoding scheme for encoding arbitrarily structured binary data. It is the main encoding method used to serialize objects for the Ethereum blockchain and cryptocurrency.

The input of the RLP encoding method is an *item*, which is either

- A *byte array*, or
- A *list of items*

RLP encoding is defined as follows:

- Case 1: For a byte array of length one with its element in the $[0x00, 0x7f (127)]$ range, this byte array is its own RLP encoding.
- Case 2: For a byte array 0-55 bytes long, the RLP encoding consists of a single byte with value $0x80 (128)$ plus the length of the byte array, followed by the original bytes in the byte array.
- Case 3: For a byte array more than 55 bytes long, the RLP encoding consists of a single byte with value $0xb7 (183)$ plus the byte length of the byte array length, followed by the length of the byte array in binary form (without leading zeros), followed by the original bytes in the byte array.

- Case 4: For a list with total length (i.e. the combined length of all its items) 0-55 bytes long, the RLP encoding consists of a single byte with value 0xc0 (192) plus the total length followed by the concatenation of the RLP encodings of the items.
- Case 5: For a list with total length more than 55 bytes long, the RLP encoding consists of a single byte with value 0xf7 (247) plus the byte length of the total length, followed by the total length in binary form, followed by the concatenation of the RLP encodings of the items.

The Java code in this example comes from the *ethereumj* project in github. We will be mainly focusing on some encoding/decoding methods in the Java class

```
ethereumj-core/src/main/java/org/ethereum/util/RLP.java
```

4.2.3.5 RLP Encoding/Decoding a Byte

The `encodeByte()` method is used to encode one single byte using RLP. It is a special case of RLP encoding when the input is a byte array of at most length 1, representing the single input byte. Note that in Ethereum integers must be represented in big endian binary form with no leading zeroes. So the byte value zero is equivalent to the empty byte array.

```
public static byte[] encodeByte(byte singleByte) {
    if ((singleByte & 0xFF) == 0) {
        return new byte[]{(byte) OFFSET_SHORT_ITEM};
    } else if ((singleByte & 0xFF) <= 0x7F) {
        return new byte[]{singleByte};
    } else {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 1), singleByte};
    }
}
```

Given a byte array containing a RLP encoded byte at the given index, the `decodeByte()` method returns the original byte. (Without loss of generality, we assumed that the index is 0 when lifting this decoding method into logic.)

```
private static byte decodeOneByteItem(byte[] data, int index) {
    // null item
    if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM) {
        return (byte) (data[index] - OFFSET_SHORT_ITEM);
    }
    // single byte item
    if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {
        return data[index];
    }
    // single byte item
    if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM + 1) {
        return data[index + 1];
    }
}
```

```

    return 0;
}

```

The next method, `decodeEncodedByte`, is written by us for the sole purpose of querying whether the RLP decoding of a RLP encoded byte returns the original byte.

```

public static byte decodeEncodedByte(byte singleByte) {
    return decodeOneByteItem(encodeByte(singleByte), 0);
}

```

Here are some of the queries we ran on the above methods:

- *Query:* If $0 \leq \text{singleByte} \leq 127$, is $\text{encodeByte}(\text{singleByte}) = [\text{singleByte}]$ always true?

Result: No, found a counter example: $\text{singleByte} = 0$ with RLP encoding `[128]`. As explained earlier, in Ethereum a byte 0 is equivalent to an empty byte array. So its RLP encoding is `[128]` instead of `[0]`.

```

(must-fail
  (prove-with-tactics '(implies (unsigned-byte-p 7 singleByte)
                                (equal (list singleByte)
                                       (encodeByte singleByte)))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If $0 < \text{singleByte} \leq 127$, is $\text{encodeByte}(\text{singleByte}) = [\text{singleByte}]$ always true?

Result: Yes, this is proved to be true.

```

(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p 7 singleByte)
                                    (not (equal 0 singleByte)))
                                (equal (list singleByte)
                                       (encodeByte singleByte)))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If singleByte is a byte, is the length of its RLP encoding always 1?

Result: No, found a counter example: $\text{singleByte} = 128$ with RLP encoding = `[129 128]`.

```

(must-fail
  (prove-with-tactics '(implies (unsigned-byte-p 8 singleByte)
                                (equal (len (encodeByte singleByte)) 1))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If singleByte is a byte, is the length of its RLP encoding either 1 or 2?

Result: Yes, this is proved to be true.

```

(must-succeed

```

```

(prove-with-tactics '(implies (and (unsigned-byte-p 8 singleByte)
                                   (equal encoded-len
                                           (len (encodeByte singleByte))))
                        (or (equal encoded-len 1) (equal encoded-len 2))))
:tactics '(:rewrite :stp)
:rules (...))

```

- *Query:* If *singleByte* is a byte, can its RLP encoding be [0]?

Result: No, it can be proved that the RLP encoding of a byte can never be [0].

```

(must-succeed
 (prove-with-tactics '(implies (and (unsigned-byte-p 8 singleByte)
                                   (equal encoded-len
                                           (len (encodeByte singleByte))))
                        (or (equal encoded-len 1) (equal encoded-len 2))))
:tactics '(:rewrite :stp)
:rules (...))

```

- *Query:* If the first byte of an RLP encoded byte is greater than 127, is the next byte always the original byte?

Result: No, found counter example: data = [128 1]. When the first byte of a RLP encoded byte is 128, the RLP encoding has length 1 and the original byte is 0.

```

(must-fail
 (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 data)
                                   (true-listp data)
                                   (equal 2 (len data))
                                   (bvlt 8 127 (bv-array-read 8 2 0 data)))
                        (equal (bv-array-read 8 2 1 data)
                               (loghead 8 (decodeOneByteItem data))))
:tactics '(:rewrite :stp)
:rules (...))

```

- *Query:* Is the RLP decoding of an RLP encoded byte always the original byte?

Result: Yes, this is proved to be true.

```

(must-succeed
 (prove-with-tactics '(equal (decodeencodedbyte singlebyte)
                             (bvsx 32 8 (loghead 8 singlebyte))))
:tactics '(:rewrite :stp)
:rules (...))
))

```


4.2.3.6 RLP Encoding/Decoding an Integer

The `encodeInt()` method is used to encode one single integer using RLP. It is a special case of RLP encoding when the input is a byte array of at most length 4, representing the single input integer. Note that in Ethereum integers must be represented in big endian binary form with no leading zeroes. So the integer value zero is encoded as the empty byte array.

```
public static byte[] encodeInt(int singleInt) {
    if ((singleInt & 0xFF) == singleInt)
        return encodeByte((byte) singleInt);
    else if ((singleInt & 0xFFFF) == singleInt)
        return encodeShort((short) singleInt);
    else if ((singleInt & 0xFFFFFFFF) == singleInt)
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 3),
            (byte) (singleInt >>> 16),
            (byte) (singleInt >>> 8),
            (byte) singleInt};
    else {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 4),
            (byte) (singleInt >>> 24),
            (byte) (singleInt >>> 16),
            (byte) (singleInt >>> 8),
            (byte) singleInt};
    }
}
```

Given a byte array containing a RLP encoded integer at the given index, the `decodeInt()` method returns the original integer. (Without loss of generality, we assumed that the index is 0 when lifting this decoding method into logic.)

```
public static int decodeInt(byte[] data, int index) {
    int value = 0;
    // NOTE: there are two ways zero can be encoded - 0x00 and OFFSET_SHORT_ITEM

    if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {
        return data[index];
    } else if ((data[index] & 0xFF) >= OFFSET_SHORT_ITEM
        && (data[index] & 0xFF) < OFFSET_LONG_ITEM) {

        byte length = (byte) (data[index] - OFFSET_SHORT_ITEM);
        byte pow = (byte) (length - 1);
        for (int i = 1; i <= length; ++i) {
            value += (data[index + i] & 0xFF) << (8 * pow);
            pow--;
        }
    } else {
```

```

        throw new RuntimeException("wrong decode attempt");
    }
    return value;
}

```

The next method, `decodeEncodedInt`, is written by us for the sole purpose of querying whether the RLP decoding of a RLP encoded byte returns the original byte.

```

public static int decodeEncodedInt(int singleInt) {
    return decodeInt(encodeInt(singleInt), 0);
}

```

Here are some of the queries we ran on the above methods:

- *Query:* If $0 \leq \text{singleInt} \leq 127$, is $\text{encodeInt}(\text{singleInt}) = [\text{singleInt}]$ always true?

Result: No, found a counter example: `singleInt = 0` with RLP encoding `[128]`. As explained earlier, in Ethereum an integer 0 is treated as an empty byte array. So its RLP encoding is `[128]` instead of `[0]`.

```

(must-fail
  (prove-with-tactics '(implies (unsigned-byte-p 7 singleInt)
                                (equal (list singleInt)
                                        (encodeInt singleInt)))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If $0 < \text{singleInt} \leq 127$, is $\text{encodeInt}(\text{singleInt}) = [\text{singleInt}]$ always true?

Result: Yes, this is proved to be true.

```

(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p 7 singleInt)
                                    (not (equal 0 singleInt)))
                                (equal (list singleInt)
                                        (encodeInt singleInt)))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If $128 \leq \text{singleInt} \leq 255$, is $\text{encodeInt}(\text{singleInt}) = [129 \text{ singleInt}]$ always true?

Result: Yes, this is proved to be true.

```

(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p 8 singleInt)
                                    (bvlt 8 127 singleInt))
                                (equal (bv-array-write 8 2 1 singleInt '(129 0))
                                        (encodeInt singleInt)))
    :tactics '(:rewrite :stp)
    :rules (...)))

```

- *Query:* If *singleInt* is an integer, is the length of its RLP encoding between 1 and 5?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p 32 singleInt)
                                     (equal encoded-len
                                             (len (encodeInt singleInt))))
                      (or (bvlt 32 0 encoded-len)
                          (bvlt 32 encoded-len 6))))
  :tactics '(:rewrite :stp)
  :rules (...))
```

- *Query:* If *singleInt* is an integer, can its RLP encoding be [0]?

Result: No, it can be proved that the RLP encoding of an integer can never be [0].

```
(must-succeed
  (prove-with-tactics '(implies (unsigned-byte-p 32 singleInt)
                              (not (equal '(0) (encodeInt singleInt))))
  :tactics '(:rewrite :stp)
  :rules (...))
```

- *Query:* If the first byte of an RLP encoded integer is 130, do the next two bytes contain the big endian binary form of the original integer?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 data)
                                     (true-listp data)
                                     (equal 5 (len data))
                                     (equal 130 (bv-array-read 8 5 0 data)))
                      (equal (decodeInt data)
                              (bvplus 32
                                     (bvmult 32 256 (bv-array-read 8 5 1
                                                                (bv-array-read 8 5 2 data))))))
  :tactics '(:rewrite :stp)
  :rules (...))
```

```
(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 data)
                                     (true-listp data)
                                     (equal 5 (len data))
                                     (bvlt 8 (bv-array-read 8 5 0 data) 128))
                      (equal (bv-array-read 8 5 0 data) (decodeInt data)))
  :tactics '(:rewrite :stp)
  :rules (...))
```

- *Query:* Is the RLP decoding of an RLP encoded integer always the original integer?

Result: Yes, this is proved to be true.

```
(must-succeed
 (prove-with-tactics '(implies (unsigned-byte-p 32 singleInt)
                               (equal (decodeencodedint singleInt) singleInt))
 :tactics '(:rewrite :stp)
 :rules (...)))
```

More on the `decodeInt()` method. This method clearly is intended to decode and return a Java int, which is at most 4 bytes. However, it does not appear to limit the number of bytes read to 4. It appears to continue to add and shift.

To illustrate this, we will try to prove that when the length indicator shows 8 bytes of data, the result returned is

```
(data[1]+data[5]) * 2^24
+ (data[2]+data[6]) * 2^16
+ (data[3]+data[7]) * 2^8
+ (data[4]+data[8])
```

- *Query:* Is the result from calling `decodeInt()` on 9 bytes of data, where the first byte is indicating 8 additional bytes, always equal to the formula above?

Result: Yes, this is proved to be true.

```
(must-succeed
 (prove-with-tactics
 '(implies (and (unsigned-byte-p-list 8 data)
                (equal 136 (bv-array-read 8 56 0 data)))
            (equal (decodeIntExtraLong data)
                   (bvplus 32
                           (bvplus 32 (shl 32 (bv-array-read 8 56 1 data) 24)
                                    (shl 32 (bv-array-read 8 56 5 data) 24))
                           (bvplus 32
                                   (bvplus 32 (shl 32 (bv-array-read 8 56 2 data)
                                                       (shl 32 (bv-array-read 8 56 6 data) 16)
                                               (bvplus 32
                                                       (bvplus 32 (shl 32 (bv-array-read 8 56 7
                                                                           (bvplus 32 (bv-array-read 8 56 4 data)
                                                                           (bv-array-read 8 56 8 data))))
                                               (bvplus 32 (bv-array-read 8 56 3 data)
                                                       (bvplus 32 (bv-array-read 8 56 8 data))))
                                   (bvplus 32 (bv-array-read 8 56 3 data)
                                           (bvplus 32 (bv-array-read 8 56 8 data))))
                           (bvplus 32 (bv-array-read 8 56 4 data)
                                   (bvplus 32 (bv-array-read 8 56 8 data))))
                   :tactics '(:rewrite :stp)
                   :rules (...)))
```

4.2.3.7 RLP Encoding/Decoding a Byte Array

The `encodeElement()` method is used to encode a byte array using RLP. It covers *Case 1* through *Case 3* of the RLP encoding defined in the beginning of Section 4.2.3.4.

```
public static byte[] encodeElement(byte[] srcData) {

    if (isNullOrZeroArray(srcData))
        return new byte[] {(byte) OFFSET_SHORT_ITEM};
    else if (isSingleZero(srcData))
        return srcData;
    else if (srcData.length == 1 && (srcData[0] & 0xFF) < 0x80) {
        return srcData;
    } else if (srcData.length < SIZE_THRESHOLD) {
        // length = 8X
        byte length = (byte) (OFFSET_SHORT_ITEM + srcData.length);
        byte[] data = Arrays.copyOf(srcData, srcData.length + 1);
        System.arraycopy(data, 0, data, 1, srcData.length);
        data[0] = length;

        return data;
    } else {
        // length of length = BX
        // prefix = [BX, [length]]
        int tmpLength = srcData.length;
        byte byteNum = 0;
        while (tmpLength != 0) {
            ++byteNum;
            tmpLength = tmpLength >> 8;
        }
        byte[] lenBytes = new byte[byteNum];
        for (int i = 0; i < byteNum; ++i) {
            lenBytes[byteNum - 1 - i] = (byte) ((srcData.length >> (8 * i)) & 0xFF);
        }
        // first byte = F7 + bytes.length
        byte[] data = Arrays.copyOf(srcData, srcData.length + 1 + byteNum);
        System.arraycopy(data, 0, data, 1 + byteNum, srcData.length);
        data[0] = (byte) (OFFSET_LONG_ITEM + byteNum);
        System.arraycopy(lenBytes, 0, data, 1, lenBytes.length);

        return data;
    }
}
```

Given a byte array containing a RLP encoded source byte array at the given index, the decode-

ByteArray() method should return the original source byte array. (Without loss of generality, we assumed that the index is 0 when lifting this decoding method into logic.)

```
private static byte[] decodeByteArray(byte[] data, int index) {
    final int length = calculateLength(data, index);
    byte[] valueBytes = new byte[length];
    System.arraycopy(data, index, valueBytes, 0, length);
    return valueBytes;
}
```

However the *ethereumj* code above and the method *calculateLength()* it calls contain bugs, causing the decoding method to return incorrect results. We discovered the problem after running queries on the decoding method. So we wrote the method below for decoding byte arrays.

```
private static byte[] decodeByteArrayNew(byte[] data, int index) {
    if ((data[index] & 0xFF) > OFFSET_LONG_ITEM
        && (data[index] & 0xFF) < OFFSET_SHORT_LIST) {
        byte lengthOfLength = (byte) (data[index] - OFFSET_LONG_ITEM);
        int length = calcLengthRaw(lengthOfLength, data, index);
        byte[] valueBytes = new byte[length];
        System.arraycopy(data, index + lengthOfLength + 1, valueBytes, 0, length);
        return valueBytes;
    } else if ((data[index] & 0xFF) > OFFSET_SHORT_ITEM
        && (data[index] & 0xFF) <= OFFSET_LONG_ITEM) {
        byte length = (byte) ((data[index] & 0xFF) - OFFSET_SHORT_ITEM);
        byte[] valueBytes = new byte[length];
        System.arraycopy(data, index + 1, valueBytes, 0, length);
        return valueBytes;
    } else if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM) { // added case
        return new byte[0];
    } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) { // added case
        return new byte[] {data[index]};
    } else {
        throw new RuntimeException("wrong decode attempt");
    }
}
```

This method is written by us for the sole purpose of querying whether the RLP decoding of a RLP encoded byte array returns the original byte array.

```
public static byte[] decodeEncodedByteArray(byte[] srcData) {
    return decodeByteArrayNew(encodeElement(srcData), 0);
}
```

Here are some of the queries we ran on the above methods:

- *Query:* If *srcData* is of length 1 and $0 \leq \text{srcData}[0] \leq 127$, is its RLP encoding always equal to itself?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (equal (len srcData) 1)
                                     (bvlt 31 (bv-array-read 8 1 0 srcData) 128))
                       (equal (encodeElement srcData) srcData))
    :tactics '(:rewrite)
    :rules (...)))
```

- *Query:* If *srcData* is of length between 2 and 55, is its RLP encoding always one byte longer than it?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (bvlt 31 1 (len srcData))
                                     (bvlt 31 (len srcData) 56)
                                     (equal encoded (encodeElement srcData)))
                       (equal (len encoded) (bvplus 32 1 (len srcData))))
    :tactics '(:rewrite)
    :rules (...)))
```

- *Query:* If *srcData* is of length 55, is its RLP encoding always equal to a byte array of length 56 containing 183 followed by the contents of *srcData*?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (equal (len srcData) 55)
                                     (equal encoded (encodeElement srcData)))
                       (and (equal (len encoded) 56)
                            (equal (bv-array-read 8 56 0 encoded) 183)
                            (equal (bv-array-read 8 56 1 encoded)
                                    (bv-array-read 8 55 0 srcData))
                            ...
                            (equal (bv-array-read 8 56 55 encoded)
                                    (bv-array-read 8 55 54 srcData))))
    :tactics '(:rewrite)
    :rules (...)))
```

- *Query:* If *srcData* is of length 56, is its RLP encoding always equal to a byte array of length 58 containing 184 and 56 followed by the contents of *srcData*?

Result: Yes, this is proved to be true.

```
(must-succeed
  (prove-with-tactics '(implies (and (equal (len srcData) 56)
                                     (equal encoded (encodeElement srcData)))
                       (and (equal (len encoded) 58)
                            (equal (bv-array-read 8 58 0 encoded) 184)
                            (equal (bv-array-read 8 58 1 encoded)
                                    (bv-array-read 8 57 0 srcData))
                            ...
                            (equal (bv-array-read 8 58 57 encoded)
                                    (bv-array-read 8 57 56 srcData))))
    :tactics '(:rewrite)
    :rules (...)))
```

```

(equal (bv-array-read 8 58 1 encoded) 56)
(equal (bv-array-read 8 58 2 encoded)
       (bv-array-read 8 56 0 srcData))
...
(equal (bv-array-read 8 58 57 encoded)
       (bv-array-read 8 56 55 srcData))))
:tactics '(:rewrite)
:rules (...))

```

- *Query:* If the encoding of a source byte array is of length 56 with the first byte being 183, does the source byte array always equal to its encoding less the first byte?

Result: Yes, this is proved to be true.

```

(must-succeed
 (prove-with-tactics '(implies (and (equal (len data) 56)
                                     (equal (bv-array-read 8 56 0 data) 183)
                                     (equal decoded (decodedByteArrayNew data)))
                          (and (equal (len decoded) 55)
                               (equal (bv-array-read 8 55 0 decoded)
                                       (bv-array-read 8 56 1 data))
                               ...
                               (equal (bv-array-read 8 55 54 decoded)
                                       (bv-array-read 8 56 55 data))))
:tactics '(:rewrite)
:rules (...))

```

- *Query:* If a source byte array is of length 56, is the decoding of its encoding always the same as itself?

Result: Yes, this is proved to be true.

```

(must-succeed
 (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 srcData)
                                     (equal 56 (len srcData))
                                     (true-listp srcData))
                          (equal (decodeEncodedByteArray srcData)
                                  srcData))
:tactics '(:rewrite :stp)
:rules (...))

```

4.2.3.8 Finding the Index of the Next Element in an RLP Encoded List

In this section, we will show an example where running queries on a Java method using our tool helped us to discover a bug in the code.

Given a byte array *payload* which is the RLP encoding of a list, and the position *pos* of an encoded

element in the byte array, the method intends to return the position for the next encoded element in the list. (Without loss of generality, we assumed that the index is 0 when lifting this decoding method into logic.)

```

public static int getNextElementIndex(byte[] payload, int pos) {
    if (pos >= payload.length)
        return -1;
    if ((payload[pos] & 0xFF) >= OFFSET_LONG_LIST) {
        byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_LIST);
        int length = calcLength(lengthOfLength, payload, pos);
        return pos + lengthOfLength + length + 1;
    }
    if ((payload[pos] & 0xFF) >= OFFSET_SHORT_LIST
        && (payload[pos] & 0xFF) < OFFSET_LONG_LIST) {
        byte length = (byte) ((payload[pos] & 0xFF) - OFFSET_SHORT_LIST);
        return pos + 1 + length;
    }
    (*) if ((payload[pos] & 0xFF) >= OFFSET_LONG_ITEM
        && (payload[pos] & 0xFF) < OFFSET_SHORT_LIST) {
        byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_ITEM);
        int length = calcLength(lengthOfLength, payload, pos);
        return pos + lengthOfLength + length + 1;
    }
    if ((payload[pos] & 0xFF) > OFFSET_SHORT_ITEM
    (**) && (payload[pos] & 0xFF) < OFFSET_LONG_ITEM) {
        byte length = (byte) ((payload[pos] & 0xFF) - OFFSET_SHORT_ITEM);
        return pos + 1 + length;
    }
    if ((payload[pos] & 0xFF) == OFFSET_SHORT_ITEM) {
        return pos + 1;
    }
    if ((payload[pos] & 0xFF) < OFFSET_SHORT_ITEM) {
        return pos + 1;
    }
    return -1;
}

```

We ran the following queries on this method.

- *Query*: If the first byte of an RLP encoded list element is between 129 and 183 (both included), is the index of next RLP encoded list element always *first byte - 128 + 1* away?

Here we are considering the case when the current list element is a byte array of length between 1 and 55, covered by *Case 2* of the RLP encoding defined in the beginning of Section 4.2.3.4. In this case, the RLP encoding should be *128 + byte array length* followed by the bytes in the byte array. Thus the index of the next RLP encoded list element should indeed be *first byte - 128 + 1* away.

Result: No, found a counter example: payload = [183 0 ... 0], i.e. when the first byte is 183 the query returns false.

```
(must-fail
  (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 payload)
                                     (true-listp payload)
                                     (equal (len payload) 60)
                                     (equal e0 (bv-array-read 8 60 0 payload))
                                     (sbvlt 32 128 e0)
                                     (sbvle 32 e0 183))
                        (equal (getNextElementIndex payload)
                              (bvplus 31 1 (bvminus 31 e0 128))))))
:tactics '(:rewrite :stp)
:rules (...))
```

The result of this query is surprising. The counter example where the first byte is 183 indicates that the first list element should be of length 55. So the index of the next element should be 56 (1 + the length 55). However the next index computed by the method is actually 1. It looks like the method is handling the boundary case incorrectly. So we ran a similar query excluding the boundary case as below.

- *Query:* If the first byte of an RLP encoded list element is between 129 and 182 (both included), is the index of next RLP encoded list element always *first byte - 128 + 1* away?

Result: Yes, this is proved to be true

```
(must-succeed
  (prove-with-tactics '(implies (and (unsigned-byte-p-list 8 payload)
                                     (true-listp payload)
                                     (equal (len payload) 60)
                                     (equal e0 (bv-array-read 8 60 0 payload))
                                     (sbvlt 32 128 e0)
                                     (sbvle 32 e0 182))
                        (equal (getNextElementIndex payload)
                              (bvplus 31 1 (bvminus 31 e0 128))))))
:tactics '(:rewrite :stp)
:rules (...))
```

So indeed the method failed to correctly handle the boundary case when the original element length is 55. When we examined the code, we found in the lines marked with (*) and (**) that the boundary case when ((payload[pos] & 0xFF) == OFFSET_LONG_ITEM) i.e. (*first byte = 183*) was treated as a *Case 3* instead of *Case 2* in the RLP encoding defined in the beginning of Section 4.2.3.4.

Similar to this example, we also discovered that the decoding method decodeByteArray() (in Section 4.2.3.7) is incorrect through queries. The bugs in that method include both boundary case mishandling and logical errors.

4.2.3.9 Ethereum bugs and pull requests

During our investigation of RLP and ethereumj, we identified 29 bugs in RLP as well as one serious bug that caused an ethereumj client node to get a fatal error at block 3943608 when doing a full sync. The ethereumj community was working around the bug by doing fast sync past that block. Fast sync just looks at the final data for each block rather than verifying the entire block, so it was a suboptimal situation.

We fixed these 30 bugs. All tests passed and the blockchain synced up to the most recent block. We submitted a pull request to github containing the 30 fixes.

4.2.3.10 Example: Base58 Encoding

Base58 is a binary-to-text encoding scheme developed for use in Bitcoin and many other cryptocurrencies. It encodes long numbers using mixed-alphanumeric representation with a base (or radix) which is 58. The alphabet in Base58 encoding includes the upper- and lowercase letters and numbers, but omits some characters, i.e. 0 (number zero), O (capital o), I (capital i) and l (lower case L), that are frequently mistaken for one another and can appear identical when displayed in certain fonts. The Base58 encoding offers a balance between compact representation, readability, and error detection and prevention.

The Java code in this example comes from the 'bitcoinj' project in github. We will be focusing on the encode() method in the Java class

```
bitcoinj/core/src/main/java/org/bitcoinj/core/Base58.java

public static final char[] ALPHABET
    = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz".toCharArray();
private static final char ENCODED_ZERO = ALPHABET[0];

/**
 * Encodes the given bytes as a base58 string (no checksum is appended).
 *
 * @param input the bytes to encode
 * @return the base58-encoded string
 */
public static String encode(byte[] input) {
    if (input.length == 0) {
        return "";
    }
    // Count leading zeros.
    int zeros = 0;
    while (zeros < input.length && input[zeros] == 0) {
        ++zeros;
    }
    // Convert base-256 digits to base-58 digits (plus conversion to ASCII
```

```

    // characters)
    input = Arrays.copyOf(input, input.length); // since we modify it in-place
    char[] encoded = new char[input.length * 2]; // upper bound
    int outputStart = encoded.length;
    for (int inputStart = zeros; inputStart < input.length; ) {
        encoded[--outputStart] = ALPHABET[divmod(input, inputStart, 256, 58)];
        if (input[inputStart] == 0) {
            ++inputStart; // optimization - skip leading zeros
        }
    }
    // Preserve exactly as many leading encoded zeros in output as there
    // were leading zeros in input.
    while (outputStart < encoded.length && encoded[outputStart] == ENCODED_ZERO) {
        ++outputStart;
    }
    while (--zeros >= 0) {
        encoded[--outputStart] = ENCODED_ZERO;
    }
    // Return encoded string (including encoded leading zeros).
    return new String(encoded, outputStart, encoded.length - outputStart);
}

/**
 * Divides a number, represented as an array of bytes each containing a single digit
 * in the specified base, by the given divisor. The given number is modified in-place
 * to contain the quotient, and the return value is the remainder.
 *
 * @param number the number to divide
 * @param firstDigit the index within the array of the first non-zero digit
 *     (this is used for optimization by skipping the leading zeros)
 * @param base the base in which the number's digits are represented (up to 256)
 * @param divisor the number to divide by (up to 256)
 * @return the remainder of the division operation
 */
private static byte divmod(byte[] number, int firstDigit, int base, int divisor) {
    // this is just long division which accounts for the base of the input digits
    int remainder = 0;
    for (int i = firstDigit; i < number.length; i++) {
        int digit = (int) number[i] & 0xFF;
        int temp = remainder * base + digit;
        number[i] = (byte) (temp / divisor);
        remainder = temp % divisor;
    }
    return (byte) remainder;
}

```

Specializations After a Java method is lifted into logic, we can specialize the resulting logical specification in various ways using APT and perform scenario-based queries on these specialized specifications. Given assumptions from the user about the specific scenario of interest, we apply the ‘simplify’ transformation with these assumptions. The transformation will propagate the assumptions through the extracted specification to simplify it and clarify it (e.g., by resolving values to particular constants and eliminating functionality that is unreachable in the particular case of interest). The result will be easier to comprehend than the initial extracted specification because it will cover only one program-specific scenario.

Here are a few specializations we performed on the Base58.encode() example. Throughout, the various encode-loop functions represent the lifted versions of the 5 loops in the program.

4.2.3.11 Specialization: Input is a Singleton Array

To specialize the lifted specification when the input contains a single element, we applied the following ‘simplify’ transformation:

```
(simplify encode
  :new-name encode-singleton
  :assumptions ((equal 1 (len input))))
```

We then applied the ‘letify’ transformation and renamed the variables to get the following more readable specialized spec:

```
(defun encode-singleton (input)
  (let* ((zeros (nth 0 (encode-loop-1 0 input 1)))
        (loop-2-result (encode-loop-2 zeros 2 input '(0 0) 1)))
    (let ((output-start (nth 1 loop-2-result))
          (encoded (nth 3 loop-2-result)))
      (let ((loop-5-result (encode-loop-5 (nth 0 (encode-loop-4 output-start encoded 1)
                                          zeros encoded 1)))
            (subrange (nth 0 loop-5-result) ;; output-start
                       1
                       (nth 2 loop-5-result)))))) ;; encoded
```

4.2.3.12 Specialization: Input Has No Leading Zeros

To specialize the lifted specification when the input contains no leading zeros, we applied the following ‘simplify’ transformation:

```
(simplify encode
  :new-name encode-no-leading-zeros
  :assumptions ((not (equal 0 (BV-ARRAY-READ 8 (len INPUT) 0 INPUT))))
```

We then applied the ‘letify’ transformation and renamed the variables to get the following more readable specialized spec:

```

(defun encode-no-leading-zeros (input)
  (let* ((input-len (len input))
         (encoded-len (bvcats 30 input-len 1 0))
         (loop-2-result (encode-loop-2 0 encoded-len input
                                     (repeat encoded-len 0) input-len))
         (encoded (nth 3 loop-2-result)))
    (subrange (nth 0 (encode-loop-4 (nth 1 loop-2-result)
                                   encoded input-len)) ;; output-start
              (bvplus 32 *minus-1* encoded-len) ;; encoded-len - 1
              encoded)))

```

Note that the resulting spec contains no calls to `encode-loop-1` and to `encode-loop-5` since these loops are eliminated under the assumption.

4.2.3.13 Specialization: Input Contains All Zeros

To specialize the lifted specification when the input contains all zeros, we applied the following ‘simplify’ transformation:

```

(simplify encode
  :new-name encode-all-zeros
  :assumptions ((unsigned-byte-p 30 (len input))
                (not (equal 0 (len input)))
                (equal (nth 0 (encode-loop-1$2 0 input (len input))
                          (len input))))

```

We then applied the ‘letify’ transformation and renamed the variables to get the following more readable specialized spec:

```

(defun encode-all-zeros (input)
  (let* ((input-len (len input))
         (encoded-len (bvcats 30 input-len 1 0)) ;; input-len * 2
         (loop-5-result (encode-loop-5 encoded-len input-len
                                       (repeat encoded-len 0) input-len))
         (subrange (nth 0 loop-5-result) ;; output-start
                    (bvplus 30 (bvplus 30 *minus-1* input-len) 1 1) ;; input-len * 2 - 1
                    (nth 2 loop-5-result)))) ;; encoded

```

Note that the resulting spec contains no calls to `encode-loop-1` through `encode-loop-4` since these loops are eliminated under the assumption.

In each case, the simplified specification for the particular case is significantly shorter than the full specification, because irrelevant functionality has been removed. We are currently working on a code generator that will be able to transform the simplified specifications back to Java code, to further enhance readability. Also, since the specifications are expressed in the logic of the theorem prover, various tools can now be applied to analyze them further (including query tools, proof tools, concrete execution, and further transformations).

4.3 String Subcomponent and Value Analysis Evaluation

Sansa was evaluated over several test suites. These include:

- **StoneSoup phase 1 tests**
- **StoneSoup phase 3 tests**
- **Amnesia SQL test suite**

The following subsections detail the results from each suite

4.3.1 StoneSoup phase 1 tests

The IARPA project 'Securely Taking On New Executable Software of Uncertain Provenance' (StoneSoup) developed and evaluated run-time protections for a variety of vulnerabilities including SQL injection.

In phase 1 of the project, the evaluation team developed 13 small SQL programs (see figure 1). Each of these programs contained one SQL statement that was vulnerable to an injection attack and a number of other SQL statements that were not vulnerable.

These programs were designed to test run-time checking systems and included both benign and attack inputs that would exercise the vulnerable code.

Sansa found each vulnerable statement with no false positives. In total, Sansa found 18 vulnerable statements and marked 197 statements as safe. In these results, each unique call chain is treated as a separate statement.

4.3.2 StoneSoup phase 3 tests

In phase 3 of the StoneSoup project, the evaluation team developed a number of different SQL vulnerabilities that they could automatically insert into existing programs [12]. These vulnerabilities were created over different sources of data (Taint Source) and several different complexity groups. (Data Type Complexity, Data Flow Complexity, and Control Flow complexity). A particular test case is constructed by choosing one feature from each group. We tested Sansa on each feature from each group. We did not test every combination of settings. We used the default program (JTree) as the test program.

The subsections below describe the features for each group. Much of this information is taken from the StoneSoup Test Data Generation Plan [12].

4.3.2.1 Taint Sources

Source taint features represent the different channels that can be used to introduce malicious input into an application. In particular, the focus is on input that could be supplied by, or manipulated by,

Test Program	Lines of code	Database	Vulnerable SQL Stmts	Safe SQL Stmts
TC-0000	1055	MySQL	2	15
TC-3008	1723	MySQL	1	17
TC-3010	1680	MySQL	1	17
TC-3014	1166	MySQL	1	17
TC-3015	1127	MySQL	1	17
TC-3016	1055	MySQL	2	15
TC-3017	1780	MySQL	3	28
TC-3044	1054	MySQL	2	15
TC-3045	1730	Postgres	1	19
TC-3166	1221	MySQL	1	17
TC-3174	1298	MySQL	1	19
TC-3177	370	MySQL	1	1
TC-3178	315	MySQL	1	0

Table 1: SQL injection tests from StoneSoup Phase 1. Each test has a single vulnerable SQL statement and other safe SQL statements. Each unique call chain is counted separately. The StoneSoup Phase 1 tests also included five tests that used the Hibernate Database. These were not included as Sansa does not support Hibernate

an attacker (and thereby considered tainted). Additionally the issue of structured versus unstructured data complicates the issue of how data might be introduced to the application being tested. Structured data lends itself to be easily validated thus making tainting it much more difficult and restricting the taint possibilities. Unstructured data because of its disorganized nature, however, would require special processing by the testing application to deal with it. It is the intention that the specific source taint be involved in the flow of code execution from the point that the data is read to the point where the vulnerable code is executed.

The following taint sources were included in the tests:

- **Environment Variable.** The source of the untrusted data is an environment variable. Normally this is obtained via the Java library method `System.getenv()`
- **File Contents.** The source of the untrusted data is a file on the local file system.
- **Socket.** The source of the untrusted data is a socket (either client or server).

4.3.2.2 Data Type Features

Data within an application is stored in memory using a variety of different constructs and structures. A variety of these different structures are used to generate the tests. These are:

- **Array.** Source data is stored and retrieved from an array. For example:

```
int a = getValue();
int [] x = new int[a];
```



```
x[0] = getInput(); // read untrusted data
exec_sql (x[0]) // use untrusted data in an \gls{sql} stmt
```

- **Simple.** Source data is stored in a built-in data type (e.g., int, float, double, etc).
- **Void Pointer.** The feature involves a generic pointer that can be used to refer to objects of any data type. In Java this uses a reference to `java.lang.Object`.

```
int x = getInput(); // read untrusted data
Object y = new Integer(x);
int z = (Integer)y;
exec_sql (z); // use untrusted data in an \gls{sql}
stmt
```

4.3.2.3 Control Flow Features

Untrusted inputs often flows through many different types and levels of control flow. Control flow refers to the order in which individual instructions are executed.

The following control flow features are included in the tests:

- **Callback.** This control feature involves passing a reference to a function or procedure such that the referenced function can be called as necessary. In Java this is implemented using interfaces.

```
/**
 * Define a simple interface that declares the method we wish
 * to be invoked when something interesting happens.
 */
public interface InterestingEvent {
    public void interestingEvent ();
}

/**
 * The code that wishes to receive the event notification must
 * implement the InterestingEvent interface and just pass a
 * reference to itself to the main event notifier.
 */
public class CallMe implements InterestingEvent {
    public CallMe () {
        // Create the event notifier and pass ourself to it.
        new EventNotifier (this);
    }

    // Define the actual handler for the event.
    public void interestingEvent () {
        // Wow! Something really interesting must have occurred!
        // STONESOUP: TRIGGER_POINT
    }
}
```

```

/**
 * This class sets up the event notification to call the method
 * we wish to invoke when something interesting happens.
 */
public class EventNotifier {
    private InterestingEvent ie;
    private boolean somethingHappened;
    public EventNotifier (InterestingEvent event) {
        ie = event;
        somethingHappened = false;
    }
    public void doWork () {
        if (somethingHappened) {
            // Signal the event by invoking the interface's method.
            ie.interestingEvent ();
        }
    }
}

/** The main function. */
public static void main(String[] args) {
    EventNotifier event = new EventNotifier(new CallMe());
    event.doWork();
}

```

- **Indirectly Recursive.** This control flow feature involves a function invocation that indirectly invokes itself recursively through one or more levels of indirection. The untrusted data is read before the recursive call and the vulnerable statement is executed within the recursive call. For example:

```

int x = getInput(); // read untrusted input
foo(x);

int foo(int y) {
    y--;
    if (y > 0) bar(y);
}

int bar(int z) {
    foo(z);
    exec_sql();
}

```

- **Infinite Loop.** This control flow feature involves an infinite loop that is executed until a break condition is met. The untrusted data is read before the loop starts and the vulnerable statement is after the loop. For example:

```

int x = getInput(); // read untrusted input
while(true) {
    if (condition == TRUE) break;
}

exec_sql(x);

```

- **Interclass 1.** This control flow feature involves a branch to a subroutine located in a different class. This is considered one level of branching. The untrusted data is read in one class and the vulnerable statement is in the second class. For example:

```
class OuterClass {
    //...
    class NestedClass {
        public static int foo(int x) {
            exec_sql (x);
        }
    }

    public static void main(String[] args) {
        NestedClass nestedObject = new NestedClass();
        int x = getInput(); // read untrusted data
        nestedObject.foo(x);
    }
}
```

- **Interclass 2.** This control flow feature involves a branch to a subroutine located in a different class that then branches to another subroutine located in another class. This is considered two levels of branching. The untrusted data is read in one class, control is passed through a second class, and the vulnerable statement is in a third class.
- **Interclass 10.** This is the same as Interclass 2 except that ten classes are involved.
- **Interclass 50.** This is the same as Interclass 2 except that fifty classes are involved.
- **Interprocedural 1.** This control flow feature involves a branch to a subroutine within the same file or class. The untrusted data is read before the subroutine call and the vulnerable statement is in the subroutine call. For example:

```
int main() {
    int taint = getInput() // read untrusted data
    func(taint);
}

void func(int taint) {
    exec_sql (taint);
}
```

- **Interprocedural 2.** This control flow feature involves a branch through two different subroutines in the same file or class. The untrusted data is read before the call to the first subroutine, the first subroutine calls a second subroutine, and the vulnerable statement is in the second subroutine. For example:

```
int main() {
    int taint = getInput() // read untrusted data
    func(taint);
}

void func(int taint) {
    func2(taint);
}
```

```

}

void func2(int taint) {
    exec-sql (taint);
}

```

- **Interprocedural 10.** This is the same as Interprocedural 2 except that ten function calls are involved.
- **Interprocedural 50.** This is the same as Interprocedural 2 except that fifty function calls are involved.
- **Interrupt.** This control flow feature is a structure where control is passed to a given block of code whenever a certain interrupt condition is met. A try/catch block and a signal handler are typical examples. The untrusted data is read before the interrupt and the vulnerable statement is in the interrupt block. For example:

```

try {
    x = getInput() // read untrusted input
    ...
}
catch(Exception e) {
    exec_sql (x);
}

```

- **Interrupt Continue.** This control flow feature is a structure where control is passed to a given block of code whenever a certain interrupt condition is met; however, an additional block of code is guaranteed to be executed regardless if the interrupt condition was met. A try/catch/finally block is an example of this feature. The untrusted data is read before the interrupt and the vulnerable statement is executed inside the guaranteed block. For example:

```

try {
    x = getInput(); // read untrusted input
    ...
}
catch (Exception e) { ... }
finally {
    exec_sql (x);
}

```

- **Recursive.** This control feature is a function that directly invokes itself recursively. The untrusted input is read before the recursive call and the vulnerable statement is within the recursive call. For example:

```

public void main() {
    x = getInput(); // read untrusted input
    m1 (x);
}

public void m1 (int x) {

```

```

if (x>5) {
    exec_sql (x);
    return;
}
m1 (++x);
}

```

- **Sequence.** This control flow feature the most basic control flow feature where statements are executed one after the other without branching or looping. The untrusted input and the vulnerable statement are within the same block of code.
- **Break with label.** This control flow feature uses a labeled break to jump out of a control flow looping structure and jump to another point within the program. The untrusted input is read before the loop starts and the vulnerable statement is after the loop. For example:

```

x = getInput() // read untrusted input
foo:
    for(int i = 0; i < 10; i++)
        for(int j = 0; j < 10; j++)
            if( bar() == true)
                break foo;
exec_sql (x);

```

- **Function invocation overloading.** This control flow feature is a function invocation that includes multiple definitions of a function. This is similar to providing default values for certain parameters. The untrusted input is provided as one of the default values and the vulnerable statement is inside the function. For example:

```

x = getInput(); // read untrusted input
foo (x);

public void foo() {
    this.z = 10;
}

public void foo(int y) {
    this.z = y;
    exec_sql (z);
}

```

4.3.2.4 Data Flow Features

Untrusted data may flow through many different variables, structures, arrays, etc before reaching a vulnerable statement. Data flow features are different ways in which the code passes a source input through the program.

The following data flow features are included in the tests:

- **Address Aliasing.** This data flow feature involves one level of indirection in accessing untrusted inputs. The untrusted data is stored in one variable and then referenced at the vulnerable statement through a different reference. For example:

```
String s = getInput(); // read untrusted input
Object o = s;
exec_sql ((String) o);
```

- **Address as a constant.** This data flow feature involves addressing the source input as a constant. The untrusted data is stored in a constant (final variable) and then used in the vulnerable statement. For example:

```
final String s = getInput(); // read untrusted input
exec_sql (s);
```

- **Address as a function return value.** This data flow feature involves using the return value of a function as the address of a value. The untrusted data is returned by the function and then used in a vulnerable statement. For example:

```
public String m1() {
    String s = getInput(); // read untrusted input
    return s;
}
String s = m1();
exec_sql (s);
```

- **Index Alias 1.** This data flow feature involves an array index where the index is a variable to which the value of a second variable is assigned. The untrusted value should be stored in an array and then referenced as an array element in the vulnerable statement. For example:

```
int z = 5;
int y = z;
int x = y;
int[] a = new String[10];
a[x] = getInput(); // read untrusted data
String s = a[y];
exec_sql (s);
```

- **Basic.** This data flow feature represents the most basic data flow feature where statements are executed one after another without aliasing or passing of the data structure. The untrusted data is read into a variable and then used within the same scope.
- **Variable argument list.** This data flow feature involves passing data into a function that utilizes a variable number of arguments. The untrusted data is read before the function call and is then passed as one of the variable arguments. The vulnerable statement is in the function that accepts a variable argument list. For example:

```
String s = getInput(); // read untrusted data
m1(3, null, null, null, untrusted_input, null, null);

static public void m1(int index, Object... arr) {
```

```

Object obj = null;
int ii = 0;
for (ii = 0; ii < arr.length; ii++) {
    if (ii == index)
        obj = arr[ii];
}
exec_sql (obj);
}

```

- **Java generics.** This data flow variant involves using Java generics within the test. The untrusted data is placed in an object that is defined with generics and then extracted from that object before being used in a vulnerable statement. For example:

```

List<String> v = new ArrayList<String>();
String s = getInput(); // read untrusted data
v.add(s);
exec_sql(v.get(0));

```

Test	Source Taint	Data Type	Data Flow	Control Flow
00	File	Simple	Basic	Interprocedural 10
01	File	Simple	Basic	Infinite loop
02	Env Var	Reference	Java generics	Recursive
03	Socket	Array	Index alias 1	Interrupt continue
04	Socket	Simple	Address as constant	Interclass 1
05	Socket	Array	Basic	Interclass 10
06	File	Reference	Index alias 1	Interclass 50
07	Env Var	Array	Var arg list	Sequence
08	Socket	Array	Address as constant	Interrupt
09	Env Var	Array	Java generics	Callback
10	File	Simple	Java generics	Indirectly
11	Env Var	Reference	Address as constant	Break with label
12	File	Simple	Java generics	Function overload
13	Env Var	Reference	Index alias 1	Break with label
14	Socket	Reference	Function return address	Interprocedural 2
15	File	Reference	Var arg list	Interprocedural 1
16	Env Var	Reference	Function return address	Interclass 2
17	Socket	Simple	Address as constant	Interprocedural 50
18	Env Var	Array	Index alias 1	Function overload

Table 2: SQL injection tests from StoneSoup Phase 3. Each test is generated from a set of features from each category. Each feature is covered in at least one of the tests.

4.3.2.5 Test Cases

We applied Sansa to 19 test cases that covered each of the input and complexity features. The features that make up these test cases are identified in Table 2

4.3.2.6 Results

Sansa correctly identified the vulnerability in all cases except for Data Flow 'Var Args'. In these tests, there are no safe [SQL](#) statements and thus false positives cannot be measured.

In the var args case, a varargs array is passed to a function that then loops through the elements of the array until the desired element is found. It then accesses that element in the array. An example (slightly modified for readability) is below.

```
...
m1(3, null, null, null, untrusted_input, null, null);

static public void m1(int index, Object... arr) {
    Object obj = null;
    int ii = 0;
    for (ii = 0; ii < arr.length; ii++) {
        if (ii == index)
            obj = arr[ii];
    }
    exec_sql (obj);
}
```

Sansa keeps precise data for arrays that are indexed only by constants. The varargs array above is allocated and its elements are set in the call to `m1()`. Unfortunately at this time, Sansa heuristic analysis of loops does not handle this correctly and only considers the first element of the array (which is null). Thus null is passed to the [SQL](#) statement rather than the `untrusted_input`.

4.3.3 Amnesia SQL test suite

The Amnesia test suite [13, 14] has been used previously to evaluate [SQL](#) defenses. We used Bookstore which is the largest (16,959 lines) of the seven applications in the suite.

Bookstore uses the `toSQL` utility routine to check strings passed to [SQL](#) statements. A slightly simplified version of that method is below:

```
String toSQL(String value, int type) {
    if (value == null)
        return "Null";

    String param = value;

    switch (type) {
    case adText: {
        param = replace(param, "\\\"", "\\\"");
        param = param.replace("'", "\\'");
        param = "'" + param + "'";
        break;
    }
    case adNumber: {
        try {
            if (!isNumber(value) || "".equals(param))
```



```

        param = "null";
    else
        param = value;
    } catch (NumberFormatException nfe) {
        param = "null";
    }
    break;
}
}
return param;
}

```

Its usage is relatively simple. For example:

```

"select member_id, member_level from members where member_login ="
+ toSQL(sLogin, adText)
+ " and member_password=" + toSQL(sPassword, adText);

```

The `toSQL` method correctly sanitizes string inputs (those whose type is `adText`). Sansa correctly analyzes `toSQL`. This requires several analysis conditions.

First, Sansa is context sensitive and is thus able to propagate the constant type into the call. Without that constant, each of the conditions of the switch statement would have to be analyzed yielding result that are not precise enough to determine the safety of the [SQL](#) statement.

Second, Sansa prunes unexecutable code in conditionals and switch statements over known values. Thus, when type is `adText` it only considers the code in the first branch of the switch statement.

Bookstore is implemented as a JavaServer Page (JSP). It reads data from an [HTTP](#) request using `HttpServletRequest.getParameter()`. Those parameters may include common [HTTP](#) encoding such as `<`; for `<`. Bookstore removes such encodings in a utility method that reads parameters:

```

String getParam(HttpServletRequest req, String paramName) {
    String param = req.getParameter(paramName);
    param = replace(param, "&", "&");
    param = replace(param, "&lt;", "<");
    param = replace(param, "&gt;", ">");
    param = replace(param, "&lt;", "<");
    param = replace(param, "&gt;", ">");
    return param;
}

```

Sansa applies each of the `replace` operations in `getParam` to the parameters as well as the `replace` operators in `toSQL`. The resulting finite automaton is relatively complex (1464 states) and took around 5 seconds to evaluate for safety. However, we modified our approach to minimize each finite automaton with each operation using the Hopcroft Algorithm [15] and the finite automaton was reduced to 72 states and the evaluation time reduced to approximately 300 milliseconds.

If the value is a number (indicated by type of `adNumber`) the `isNumber` method is called. The original method used the conversion to `Double` to check for a valid number:

```

boolean isNumber(String param) {
    boolean result;
    if (param == null || param.equals(""))
        return true;
    param = param.replace('d', '_').replace('f', '_');
    try {
        Double dbl = new Double(param);
        result = true;
    } catch (NumberFormatException nfe) {
        result = false;
    }
    return result;
}

```

Currently, Sansa analysis does not support the constructs used in this function. It does not fully support exception flows and it also doesn't support implicit flows such as the ones above that set the boolean value result. Both of these could be added. Exception control flow is straightforward. Associating the boolean with the result of `newDouble()` has several issues. First, the model of the `Double` constructor must properly constrain the value of `param` in both the normal and the exceptional conditions. Second, the value of `result` must include a constraint on `param`. This is very similar to how Sansa handles boolean functions on strings such as `contains`. See [4.1.4](#) for more information.

For the purpose of this evaluation, we replaced the check with a regular expression check.

```

boolean isNumber(String param) {
    return param.matches ("^[-+]?[0-9]*\\.?[0-9]+([eE] [-+]?[0-9]+)?$.");
}

```

We tested Bookstore on the Login entry point. There are a number of other entry points, but the same sanitization code is used in each, so we didn't check each individually. There are two [SQL](#) statements that are called within Login. The essential code for the first is:

```

...
String sLogin = getParam(request, "Login");
String sPassword = getParam(request, "Password");
ResultSet rs = stat.executeQuery (select member_id, member_level "
    + "from members where member_login =" + toSQL(sLogin, adText)
    + " and member_password=" + toSQL(sPassword, adText));

```

This call is correctly marked as safe. The strings passed to the [SQL](#) statement are correctly sanitized in `toSQL()`

The second [SQL](#) statement is

```

session.setAttribute ("UserID", rs.getString(1));
...
String UserID = dLookup(stat, "members", "member_login",
    "member_id =" + session.getAttribute("UserID").toString());

String dLookup(Statement st, String table, String fName, String where) {

```

```
ResultSet rsLookUp = st.executeQuery ("SELECT " + fName
                                     + " FROM " + table + " WHERE " + where);
}
```

In this case, the value for UserID that is passed to the [SQL](#) statement is not sanitized. The application stores it in the session and then later retrieves it. The application presumes that the value read back from the database (`rs.getString()`) is valid. If desired, Sansa can be configured to presume that information read back from the database is trusted. As currently implemented, however, it will lose track of the value within the session hash map. Sansa could be enhanced to track values within maps in the same way that it currently supports constant indices into arrays. This would be a relatively simple enhancement. Currently, though, it will issue an error on this statement indicating that the value is not a valid number.

4.3.3.1 Verifying sanitization methods

We also wanted to check and make sure that Sansa correctly found errors with incorrect versions of the sanitization method. We modified `toSQL` to modify the sanitization check and verified that an error was issued in each case.

In the first case, we removed the `replace` call that escaped quote characters. Without this check the sanitization is clearly not correct and Sansa issues an error. We also removed only the `replace` that escapes backslashes. Without that call, an attacker can create an input string of the form “. . . \’ . . .”. This will be converted to “. . . \\’ . . .”. Since the backslash is escaped, it cannot escape the single quote and the quote will prematurely terminate the application quotes and enable injected code. Sansa correctly issued an error in this situation as well. As described in subsection [4.1.8](#), Sansa evaluates the regular expression to determine if it is possible for it to include an unescaped quote.

5.0 CONCLUSION

Independently developed applications (often sold through application marketplaces) provide a plethora of capabilities at reasonable cost. It is often not possible (from a cost perspective) to develop desired applications directly. A critical weakness with purchased applications is, however, that users have no way to be sure that the applications they purchase are free of security vulnerabilities and/or malware. The potential presence of malicious malware and the resulting possibility of widespread security vulnerabilities can even eliminate the ability of service organizations (such as the United States Department of Defense) with stringent security needs to use application marketplaces.

The Sansa project developed effective program analysis techniques and tools to better understand Java programs. These techniques can statically identify vulnerabilities (such as SQL injection) and allow analysts to better understand the details of application code. Such understanding can identify possible malware in applications.

String subcomponent Taint and Value Analysis analyzes the construction of strings in a whole-program analysis. Strings are represented as finite automata. Finite automata are analyzable and can be used to check for a variety of possible vulnerabilities including SQL injection, cross-site scripting, command injection and other string related vulnerabilities. We evaluated this component against a three separate test suites and found that with a few exceptions vulnerable code was identified with very few false positives.

High-Level Functionality Queries enable a security auditor to ask questions about an application's behavior that go beyond low-level, application-independent properties (such as null pointer dereferencing). As demonstrated on a variety of examples, the toolkit can reveal concrete inputs that cause application behaviors of interest to the analyst, or prove that no such inputs exist. The toolkit can also clarify and simplify the behavior of complex code with respect to user-supplied assumptions, to allow "what if" reasoning about tricky code that is suspected of being malicious. Together, these capabilities enable more effective code audits and provide mechanized support to reason about what the code does.

6.0 REFERENCES

- [1] Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, and Sundaresan, Vijay, “Soot - a Java Bytecode Optimization Framework,” in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, CASCON '99, pp. 13–, URL <http://dl.acm.org/citation.cfm?id=781995.782008>
- [2] Møller, Anders, “dk.brics.automaton – Finite-State Automata and Regular Expressions for Java,” , 2017, <http://www.brics.dk/automaton/>
- [3] Yu, Fang, Alkhalaf, Muath, Bultan, Tevfik, and Ibarra, Oscar H., “Automata-based Symbolic String Analysis for Vulnerability Detection,” *Form. Methods Syst. Des.*, **44**, 1, February 2014, pp. 44–70, URL <http://dx.doi.org/10.1007/s10703-013-0189-1>
- [4] University of Texas at Austin, “The ACL2 Theorem Prover,” <http://www.cs.utexas.edu/~moore/acl2>
- [5] Smith, Eric W. **Axe: An Automated Formal Equivalence Checking Tool for Programs**, Ph.D. dissertation, Stanford University, 2011
- [6] Moore, J, “Proving Theorems about Java and the JVM with ACL2,” <http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html>
- [7] McCarthy, John, *A Formal Description of a Subset of Algol*, Technical Report Stanford Artificial Intelligence Project Memo No. 24, Stanford University, 1964
- [8] Manolios, Panagiotis and Moore, J Strother, “Partial Functions in ACL2,” *Journal of Automated Reasoning*, **31**, p. 2003
- [9] “APT (Automated Program Transformations): Web page,” <http://www.kestrel.edu/home/projects/apt>
- [10] Coglio, Alessandro, Kaufmann, Matt, and Smith, Eric, “A Versatile, Sound Tool for Simplifying Definitions,” in *Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, to appear
- [11] Ganesh, Vijay and Dill, David L., “A Decision Procedure for Bit-vectors and Arrays,” in *Proceedings of the 19th International Conference on Computer Aided Verification*, Springer-Verlag, Berlin, Heidelberg, CAV'07, pp. 519–531, URL <http://dl.acm.org/citation.cfm?id=1770351.1770421>

- [12] Authors, Various, “IARPA STONESOUP Documents,” https://samate.nist.gov/SRD/around.php#stonesoup_documents, URL https://samate.nist.gov/SRD/around.php#stonesoup_documents
- [13] Halfond, William G. J., Orso, Alessandro, and Manolios, Panagiotis, “Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks,” in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE 2006)*, Portland, Oregon, USA, pp. 175–185
- [14] Halfond, William G.J. and Orso, Alessandro, “AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks,” in *Proceedings of the International Conference on Automated Software Engineering*, Long Beach, California, USA, pp. 174–183
- [15] Hopcroft, John, “An $n \log n$ algorithm for minimizing states in a finite automaton,” in *Proceedings of the International Symposium on Theory of Machines and Computations*

7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACL2 A Computational Logic for Applicative Common Lisp. 3

APAC Automated Program Analysis for Cybersecurity. 4

API Application Programming Interface. 17

APT Automated Program Transformations. 18

FA finite automaton. 2, 3, 9, 10

HTTP hypertext transfer protocol. ii, 6, 62

IR intermediate representation. 5

JVM Java Virtual Machine. 15

RLP Recursive Length Prefix encoding. 4, 34

SMT Satisfiability Modulo Theories. 24

SQL Structured Query Language. i, 1, 3, 4, 6, 13, 15, 52, 60, 61, 62, 63, 64

STP Simple Theorem Prover. 23, 24