Rule-Based Static Analysis of Network Protocol Implementations

Octavian Udrea, Cristian Lumezanu, Jeffrey S. Foster

University of Maryland, College Park

Abstract

Today's software systems communicate over the Internet using standard protocols that have been heavily scrutinized, providing some assurance of resistance to malicious attacks and general robustness. However, the software that implements those protocols may still contain mistakes, and an incorrect implementation could lead to vulnerabilities even in the most well-understood protocol. The goal of this work is to close this gap by introducing a new technique for checking that a C implementation of a protocol matches its description in an RFC or similar standards document. We present a static (compile-time) source code analysis tool called Pistachio that checks C code against a rule-based specification of its behavior. Rules describe what should happen during each round of communication, and can be used to enforce constraints on ordering of operations and on data values. Our analysis is not guaranteed sound due to some heuristic approximations it makes, but has a low false negative rate in practice when compared to known bug reports. We have applied Pistachio to two different implementations of SSH2 and an implementation of RCP. Pistachio discovered a multitude of bugs, including security vulnerabilities, that we confirmed by hand and checked against each project's bug databases.

1 Introduction

Networked software systems communicate using protocols designed to provide security against attacks and robustness against network glitches. There has been a significant body of research, both formal and informal, in scrutinizing abstract protocols and proving that they meet certain reliability and safety requirements [1–5] (to name only a few). These abstract protocols, however, are ultimately implemented in software, and an incorrect implementation could

Email addresses: udrea@cs.umd.edu (Octavian Udrea), lume@cs.umd.edu (Cristian Lumezanu), jfoster@cs.umd.edu (Jeffrey S. Foster).

lead to vulnerabilities even in the most heavily-studied and well-understood protocol.

In this paper we present a tool called Pistachio that helps close this gap. Pistachio is a static (compile-time) analysis tool that checks that each communication step taken by a protocol implementation matches an abstract specification. Because it starts from a detailed protocol specification, Pistachio is able to check communication properties that generic tools such as buffer overflow detectors do not look for. Our static analysis algorithm is also very fast, enabling Pistachio to be deployed regularly during the development cycle, potentially on every compile.

The input to our system is the C source code implementing the protocol and a *rule-based* specification of its behavior, where each rule typically describes what should happen in a "round" of communication For example, the IETF current draft of the SSH connection protocol [6] specifies that "When either party wishes to terminate the channel, it sends *SSH_MSG_CHANNEL_CLOSE*. Upon receiving this message, a party *must* send back a *SSH_MSG_CHANNEL_CLOSE*. . .."

This statement translates into the following rule (slightly simplified):

`recv(_, `*in*`, _)`
in[0] = *SSH_MSG_CHANNEL_CLOSE*
⇒
`send(_, `*out*`, _)`
out[0] = *SSH_MSG_CHANNEL_CLOSE*

This rule means that after seeing a call to `recv()` whose second argument points to memory containing *SSH_MSG_CHANNEL_CLOSE*, we should reply with the same type of message. The full version of such a rule would also require that the reply contain the same channel identifier as the initial message.

In addition to this specification language, another key contribution of Pistachio is a novel static analysis algorithm for checking protocol implementations against their rule-based specification. Pistachio performs symbolic execution, based on *abstract interpretation* [7], to simulate the execution of program source code, keeping track of the state of program variables and of *ghost variables* representing abstract protocol state, such as the last value received in a communication. Using a fully automatic theorem prover, Pistachio checks that whenever it encounters a statement that triggers a rule (e.g., a call to `recv`), on all paths the conclusion of the rule is eventually satisfied (e.g., `send` is called with the right arguments). Although this seems potentially expensive, our algorithms run efficiently in practice because the code corresponding to a round of communication is relatively compact. Our static analysis is not guaranteed to find all rule violations, both because it operates on C, an unsafe

language, and because the algorithm uses some heuristics to improve performance. In practice, however, our system missed only about 5% of known bugs when measured against a bug database.

We applied Pistachio to three benchmarks: the LSH and OpenSSH implementations of SSH2 and the RCP implementation from Cygwin. Our SSH2 specification was originally developed with only LSH in mind, and we found that only two out of 96 rules needed to be adjusted to accurately cover both LSH and OpenSSH, suggesting that our approach is easily portable. Analysis took less than a minute for all of the test runs, and Pistachio detected a multitude of bugs in the implementations, including many security vulnerabilities. For example, Pistachio found a known problem in LSH that causes it to leak privileged information [8]. Pistachio also found a number of buffer overflows due to rule violations, although Pistachio does not detect arbitrary buffer overflows. We confirmed the bugs we found against bug databases for the projects, and we also found two new, unconfirmed security bugs in LSH: a buffer overflow and an incorrect authentication failure message when using public key authentication.

We categorized the rules in our specifications according to their use, and found that all of the rule categories contribute significantly to Pistachio's warnings—or, put another way, programmers make mistakes in all aspects of protocols. We also categorized the underlying code defects using a subset of Beizer's bug taxonomy [9]. We found that there were relatively few defects related to misuse of interfaces, perhaps because the network protocol implementations are spread across few modules, but that otherwise defects range over the usual space of mistakes.

Based on our results, we believe that Pistachio can be a valuable tool in ensuring the safety and security of network protocol implementations. In summary, the main contributions of this work are:

- We present a rule-based specification language for describing network protocol implementations. Using pattern matching to identify routines in the source code and ghost variables to track state, we can naturally represent the kinds of English specifications made in documents like RFCs. (Section 2)
- We describe a static analysis algorithm for checking that an implementation meets a protocol specification. Our approach uses symbolic execution to simulate the program and an automatic theorem prover to determine whether the rules are satisfied. (Section 3)
- We have applied our implementation, Pistachio, to LSH, OpenSSH, and RCP. Pistachio discovered a wide variety of known bugs, including security vulnerabilities, as well as two new, unconfirmed bugs. Overall Pistachio missed about 5% of known bugs and had a 38% false positive rate. (Section 4)

An earlier version of this work appeared in the Usenix Security conference [10]. There are several new contributions of this version of the paper. First, we extended our experimental evaluation and ran Pistachio on OpenSSH, whereas the original paper only used LSH and RCP. This allowed us to evaluate the portability of rule specifications from one version to another, as well as to compare the warning and bug distributions for the two versions of the same protocol. Second, our experimental results now uses a bug classification system, derived from Beizer [9], which allows us to categorize defects in a more systematic way and compare defects in protocol implementations to defects in general software. Third, we describe a fact substitution process that occurs during rule checking, which was omitted from the conference version, and adjusted our rule checking algorithms slightly to show rule firing more clearly. Fourth, we include a brief discussion of rule specification for library functions. Lastly, we added two new appendices, describing the language grammar and our choice of theorem prover. We also made numerous small improvements throughout the paper.

2 Rule-Based Protocol Specification

The first step in using Pistachio is developing a rule-based protocol specification, usually from a standards document. As an example, we develop a specification for a straightforward extension of the alternating bit protocol [11]. Here is a straw-man description of the protocol:

> *The protocol begins by sending the value* $n = 1$*. In each round, if* n *is received then send* $n + 1$*; otherwise resend* n*.*

Fig. 1 gives a sample C implementation of this protocol. Here `recv()` and `send()` are used to receive and send data, respectively. Notice that this implementation is actually flawed—in statement 6, `val` is incremented by 2 instead of by 1.

To check this protocol, we must first identify the communication primitives in the source code. In this case we see that the calls to `send()` in statements 2 and 7 and the call to `recv()` in statement 4 perform the communication. More specifically, we observe that we will need to track the value of the second argument in the calls, since that contains a pointer to the value that is communicated.

We use *patterns* to match these function calls or other expressions in the source code. Patterns contain *pattern variables* that specify which part of the call is of interest to the rule. For this protocol, we use pattern `send(_,` *out*`,`

```
0  int main(void) {
1    int sock, val = 1, recval;
2    send(sock, &val, sizeof(int));
3    while(1) {
4      recv(sock, &recval, sizeof(int));
5      if (recval == val)
6        val += 2;
7      send(sock, &val, sizeof(int));
8    }
9  }
```

Fig. 1. Simple alternating bit protocol implementation

_) to bind pattern variable *out* to the second argument of `send()`, and we use pattern `recv(_,` *in*`, _)` to bind *in* to the second argument of `recv()`. For other implementations we may need to use patterns that match different functions. Notice that in both of these patterns, we are already abstracting away some implementation details. For example, we do not check that the last parameter matches the size of `val`, or that the communication socket is correct, i.e., these patterns will match calls even on other sockets.

Patterns can be used to match any function calls. For example, we have found that protocol implementers often create higher-level functions that wrap send and receive operations, rather than calling low-level primitives directly. Using patterns to match these functions can make for more compact rules that are faster to check, though this is only safe if those routines are trusted.

2.1 Rule Encoding

Once we have identified the communication operations in the source code, we need to write rules that encode the steps of the protocol. Rules are of the form

$$(P_H, H) \Rightarrow (P_C, C, G)$$

where H is a *hypothesis* and C is a *conclusion*, P_H and P_C are patterns, and G is a set of assignments to ghost variables representing protocol state. In words, such a rule means: If we find a statement s in the program that matches pattern P_H, and the facts in H do not contradict the current state, then assume that H holds, and make sure that on all possible execution paths from s there is some statement matching P_C, and moreover at that point the conditions in C must hold. If a rule is satisfied in this manner, then the side effects G to ghost variables hold after the statements matching P_C. Appendix A describes the concrete grammar for rules used in our implementation.

Rule				Description
(1)		$\varepsilon \Rightarrow$	`send(_, out, _)` $out[0..3] = 1$ $n := 1$	The protocol begins by sending the value 1
(2)	`recv(_, in, _)` $in[0..3] = n$	$\Rightarrow$	`send(_, out, _)` $out[0..3] = in[0..3] + 1$ $n := out[0..3]$	If n is received then send $n + 1$
(3)	`recv(_, in, _)` $in[0..3] \neq n$	$\Rightarrow$	`send(_, out, _)` $out[0..3] = n$	Otherwise resend n

Fig. 2. Rule-based protocol specification

As an example, Fig. 2 contains the rules for our alternating bit protocol. Rule (1) is triggered by the start of the program, denoted by the special pattern ε. The hypothesis of this rule is empty, i.e., true. This rule says that on all paths from the start of the program, `send()` must be called, and its second argument must point to a 4-byte block containing the value 1. We can see that this rule is satisfied by statement 2 in Fig. 1. As a side effect, the successful conclusion of rule (1) sets the ghost variable n to 1. Thus the value of n corresponds to the data stored in `val`. Notice that there is a call to `send()` in statement 7 that could match the conclusion pattern—but it does not, because we interpret patterns in rules to always mean the *first* occurrence of a pattern on a path.

Rule (2) is triggered by a call to `recv()`. It says that if `recv()` is called, then assuming that the value n is received in the first four bytes of in, the function `send()` must eventually be called with $in + 1$ as an argument, and as a side effect the value of n is incremented. Similarly to rule (1), this rule matches the first occurrence of `send()` following `recv()`. In our example code this rule is not satisfied. Suppose rule (1) has triggered once, so the value of n is 1, and we trigger rule (2) in statement 4. Then if we assume n is received in statement 4, then statement 7 will send $n + 2$. Hence Pistachio signals a rule violation on this line.

Suppose instead that the implementation had been correct (line 6 had incremented `val` by 1), and rule (2) had succeeded. Then as a side effect of satisfying rule (2), we would set $n = 2$ on line 7 and continue iterating the body of the loop, which matches rule (2) again, this time with a different value of n. We would then need to continue iterating to check that the rule was satisfied again. Since this process could continue indefinitely, Pistachio repeats the check until it either finds a fixpoint or until it iterates a fixed maximum number of times.

Finally, rule (3) is triggered on the same call to `recv()` in statement 4. It says that if we assume the value of n is not received in in, then eventually `send()` is called with n as the argument. This rule will be satisfied by the implementation, because when we take the false branch in statement 5 we will

resend `val`, which always contains n after rules (1) or (3) fire.

2.2 Developing Rule-Based Specifications

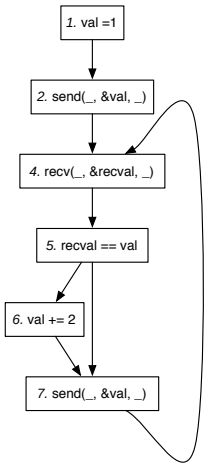
As part of our experimental evaluation (Section 4), we developed rule-based specifications for the SSH2 protocol and the RCP protocol. In both cases we started with a specification document such as an RFC or IETF standard. We then developed rules from the textual descriptions in the document, using the following three steps; Section 4.1 discusses the actual rule-based specifications we developed for our experiments.

Identifying patterns. The specification writer can either choose the low-level communication primitives as the primary patterns, as in Fig. 2, or write rules in terms of higher-level routines. We attempted both approaches when developing our specifications, but decided in favor of the first method for more future portability. This approach allowed us to use a single specification for checking both LSH and OpenSSH, which use different wrapper functions around the same low-level send and receive primitives.

Defining the rules. The main task in constructing a rule-based specification is, of course, determining the basic rules. The specification writer first needs to read the standards document carefully to discover what data is communicated and how it is represented. Then to discover the actual rules they should study the sections of the specification document that describe the protocol's behavior. We found that phrases containing *must* are good candidates for such rules. (For a discussion of terms such as *must*, *may*, and *should* in RFC documents, see RFC 2223 [12].)

For instance, as we read the SSH2 standard we learned that the message format for SSH user authentication starts with the message type, followed by the user name, service name, and method name. Furthermore, we found that the use of "none" as the authentication method is strongly discouraged by the specification except for debugging purposes. This suggested a potential security property: To prevent anonymous users from obtaining remote shells, we should ensure that *If we receive a user authentication request containing the* none *method, we must return SSH_MSG_USERAUTH_FAILURE.* Once we determine this rule, it is easy to encode it in Pistachio's notation, given our knowledge of SSH message formats.

Describing state. Finally, as we are constructing the rules, we may discover that some protocol steps are state-based. For instance, in the SSH2 protocol, any banner message has to be sent before the server sends *SSH_MSG_USERAUTH_SUCCESS.* To keep track of whether we have sent the success message, we introduce a



(a) Control-flow graph

Rule (1)	**Hyp:** facts = $\emptyset$ **Concl:** stmt 2 matches, facts = $\{val = 1\}$ **Need to show:** $\{val = 1\} \wedge \{\&val = out\} \Rightarrow out[0..3] = 1$ **Action:** $n := 1$
Rule (3)	**Hyp:** stmt 4 matches, facts = $\{n = 1, val = 1, in = \&recval, in[0..3] \neq n\} = F$ **Branch:** Since assumptions $\Rightarrow$ ($recval \neq val$), false branch taken **Concl:** stmt 7 matches, same facts F as above **Need to show:** $F \wedge \{\&val = out\} \Rightarrow out[0..3] = n$ **Action:** none
Rule (2)	**Hyp:** stmt 4 matches, facts = $\{n = 1, val = 1, in = \&recval, in[0..3] = n\}$ **Branch:** Since assumptions $\Rightarrow$ ($recval = val$), true branch taken **Concl:** stmt 7 matches, facts are $F = \{n = 1, val = 3, in = \&recval, in[0..3] = n\}$ **Need to show:** $F \wedge \{\&val = out\} \Rightarrow (out[0..3] = in[0..3] + 1)$ **Fails** to hold; issue warning

(b) Algorithm trace

Fig. 3. Static checking of example program

new ghost variable called *successSent* that is initially 0 (here we assume for simplicity that we only have one client). We modify our rules to set *successSent* to 1 or 0 in the appropriate cases. Then the condition on banner messages can be stated as *Given that successSent is 1 and for any message received, the output message type is different from SSH_MSG_USERAUTH_BANNER.* Our experience is that coming up with the ghost variables is the least-obvious part of writing a specification and requires some insight. In the SSH2 and RCP protocols, the state of the protocol usually depends on the previous message that was sent, and so our rules use ghost variables to track the last message.

3 Static Analysis of Protocol Source Code

Given a set of rules as described in Section 2 and the source code of a C program, Pistachio performs static analysis to check that the program obeys the specified rules. Pistachio uses abstract interpretation [7] to symbolically execute source code. The basic idea is to associate a set of *facts* with each point during execution. In our system, the facts we need to keep track of are the predicates in the rules and anything that might be related to them. Each statement in the program can be thought of as a *transfer function* [13], which is a "fact transformer" that takes the set of facts that hold before the statement and determines the facts that hold immediately after:

- After an assignment statement *var* = *expr*, we first remove any previous facts about *var* and then add the fact *var* = *expr*. For example, consider the code in Fig. 1 again. If before statement 6, $\{val = n\}$ is the set of facts that hold, after the assignment in statement 6 the set $\{val = n + 2\}$ holds.

Pointers and indirect assignments are handled similarly, as discussed below.

- If we see a conditional *if (p) s1 else s2*, we add the fact that p holds on the true branch, and that $\neg p$ holds on the false branch. For example, at the beginning of statement 6 in Fig. 1, we can always assume $recval = val$, since to reach this statement the condition in statement 5 must have been true.
- If we see a function call *f(x1, ..., xn)*, we propagate facts from the call site through the body of f and back to the caller. In other words, we treat the program as if all functions are inlined. As discussed below, we bound the number of times we visit recursive functions to prevent infinite looping, although recursive functions are rare in practice for network protocol implementations.

Facts in Pistachio can use most of the arithmetic and comparison expressions supported by C, including the address-of operator &. Internally, Pistachio represents all variables in memory as byte arrays, and facts can use the notation *var*$[i..j]$ to denote the number represented by bytes i through j of *var*. Pistachio facts may also use a *len* operator, which returns the length of a string variable. A detailed grammar for Pistachio rules is given in Appendix A

We perform our analysis on a standard *control-flow graph* (CFG) constructed from the program source code. In the CFG, each statement forms a node, and there is an edge from s_1 to s_2 if statement s_1 occurs immediately before statement s_2. For example, Fig. 3(a) gives the CFG for the program in Fig. 1.

Fig. 4 presents our symbolic execution algorithm more formally. The goal of this algorithm is to update *Out*, a mapping such that *Out*(s) is the set of facts that definitely hold just after statement s. The input to *FactDerivation* is an initial mapping *Out*, a set of starting statements S, and a set of ending statements T. The algorithm simulates the execution of the program from statements in S to statements in T, updating *Out* as a side effect as it executes. In this pseudocode, we write *pred*(s) and *succ*(s) for the predecessor and successor nodes of s in the CFG. Our algorithm uses an automatic theorem prover to determine which way conditional branches are taken. We write *Theorem-prover*(p) for the result of trying to prove p. This function may return *yes* if p is provably true, *no* if p is provably false, or *maybe* if the theorem prover cannot show either.

In Fig. 4, we use a worklist Q of statements, initialized on line 1 to S. We repeatedly pick statements from the worklist until it is empty. When we reach a statement in T on line 6, we stop propagation along that path. Because the set of possible facts is large (most likely infinite), simulation might not terminate if the code has a loop. Thus on line 10 we heuristically stop iterating once we have visited a statement *max_pass* times, where *max_pass* is a predetermined constant bound. Based on our experiments, we set *max_pass* to 75. We settled on this value empirically by observing that if we vary the number of iterations,

FactDerivation(*Out*, *S*, *T*)

Input:
Out — The initial facts at each program point
S — Program statements to start symbolic execution from
T — Program statements where symbolic execution ends
Output: Updates the set of facts *Out*

```
1  Q ← S
2  while Q not empty do
3      s ← dequeue(Q)
4      visit(s) ← visit(s) + 1
5      In ← ∩_{s'∈pred(s)} { Out(s') ∪ {p}   if s' is "if(p) then s else s2"
                           { Out(s') ∪ {¬p}  if s' is "if(p) then s1 else s"
                           { Out(s')         otherwise
6      if s ∈ T then
7          Out(s) ← In
8          continue
9      end if
10     if visit(s) > max_pass then
11         continue
12     end if
13     if s is assignment "var=expr" then
14         Out(s) ← (In − {facts involving var}) ∪ {var=expr}
15         Q ← Q ∪ succ(s)
16     else if s is "if(p) then s1 else s2"  then
17         Out(s) ← In
18         if Theorem-prover(Out(s) ⇒ p) = yes then
19             Q ← Q ∪ {s1}
20         else if Theorem-prover(Out(s) ⇒ ¬p) = yes then
21             Q ← Q ∪ {s2}
22         else
23             Q ← Q ∪ {s1, s2}
24         end if
25     else if s is "f(x1, ..., xn)" then
26         map ← mapping between actual and formal parameters
27         start ← entry statement of f
28         T' ← exit statements of f
29         pred(start) ← pred(start) ∪ {pred(s)}
30         Out' ← map(Out)
31         FactDerivation(Out', {start}, T')
32         Out(s) ← ∩_{s'∈T'} map^{-1}(Out'(s'))
33         Q ← Q ∪ succ(s)
34         pred(start) ← pred(start) − {pred(s)}
35     end if
36     Out(s) ← FactSubstitution(Out(s))
37 end while
```

Fig. 4. Symbolic execution algorithm

then the overall false positive and false negative rates from Pistachio rarely changed after 75 iterations in our experiments.

On line 5 we compute the set *In* of facts from the predecessors of s in the CFG. If the predecessor was a conditional, then we also add in the appropriate guard based on whether s is on the true or false branch. Then we apply a transfer function that depends on statement s. Lines 13–15 handle simple assignments, which kill and add facts as described earlier, and then add successor statements to the worklist. Lines 16–24 handle conditionals. Here we use an automatic theorem prover to prune impossible code paths. If the guard p holds in the current state, then we only add s_1 to the worklist, and if $\neg p$ holds then we only add s_2 to the worklist. If we cannot prove either, i.e., we do not know

FactSubstitution(F)

Input: F — a set of facts
Output: A superset of F containing any facts obtained from substitution

```
1  Q ← F
2  while Q ≠ ∅ do
3      f ← dequeue(Q)
4      for all variables var in f do
5          if ∃ (var = expr) ∈ (F − {f}) then
6              f′ ← replace var by expr in f
7              F ← F ∪ {f′}
8              Q ← Q ∪ {f′}
9          end if
10     end for
11     return F
12 end while
```

Fig. 5. Fact substitution algorithm

which path we will take, then we add both s_1 and s_2 to the worklist. Lines 25–35 handle function calls. We compute a renaming *map* between the actual and formal parameters of f, and then recursively simulate f from its entry node, whose predecessor is temporarily set to include *pred*(s), to its exit nodes, which are *return* statements plus the last statement in the function body. We start simulation in state *map*(Out), which contains the facts in *Out* renamed by *map*. Then the state after the call returns is the intersection of the states at all possible exits from the function, with the inverse mapping map^{-1} applied.

Finally, just before *FactDerivation* continues with the next statement on the worklist, it performs *fact substitution* on $Out(s)$ on line 36. Fig. 5 defines the fact substitution algorithm, which expands equalities. The input to the algorithm is a set of facts F. The algorithm initializes worklist Q to F on line 1. Then on lines 4–10, we iteratively remove a single fact f from Q and replace any variables *var* that occur in f according to equalities $var = expr$, if any, in the remaining facts. (Here *expr* is a complex expression, and not just a variable.) Any newly-derived facts are added back to F and Q.

Fact substitution is useful for preserving information across assignment statements. Consider the following C code, where we have listed the facts that hold without fact substitution after each line:

```
1    b = c + 1; /* {b = c + 1} */
2    a = b + 1; /* {b = c + 1,  a = b + 1} */
3    b = c + 3; /* {b = c + 3} */
```

Here since statement 3 writes to b, when computing $Out(3)$, we first killed facts about b from the incoming fact set $Out(2)$, and then added the fact generated by the assignment.

If instead we use our fact substitution algorithm, we derive the following facts after each program point:

```
1   b = c + 1; /* {b = c+1} */
2   a = b + 1; /* {b = c+1,  a = b+1,  a = (c+1)+1} */
3   b = c + 3; /* {b = c+3,  a = (c+1)+1} */
```

Here at the end of statement 2, we substituted for b in the fact $a = b+1$, producing the new fact $a = (c+1)+1$. Since this new fact does not contain b, it is not killed by the assignment in line 3. Thus *FactSubstitution* helps us preserve implied equalities even when facts are killed by assignments. Note that *FactSubstitution* is not complete in any sense—because it iterates through F only once, it may miss transitively applied equalities, and the output of *FactSubstitution* depends on the order the facts are visited in. Moreover, in pathological cases the algorithm might not terminate. However, it has proven useful in practice for improving the precision of Pistachio, and it always terminated for our experiments.

Handling other features C includes a number of language features not covered in Fig. 4. Pistachio uses CIL [14] as a front-end, which internally simplifies many C constructs by introducing temporary variables and translating loops into canonical form. We unroll loops up to *max_pass* times in an effort to improve precision. However, as discussed in Section 3.2, we attempt to find a fixpoint during unrolling process and stop if we can do so, i.e., if we can find a loop invariant.

C also includes pointers and a number of unsafe features, such as type casts and unions. Pistachio tracks facts about pointers during its simulation, and all C data is modeled as arrays of bytes with bounds information. When there is an indirect assignment through a pointer, Pistachio uses the current set of facts to determine what the pointer points to, and then updates facts about the pointed-to variable appropriately. There are two potential sources of unsoundness for such indirect updates: First, if we lose facts about a pointer when we intersect fact sets at a join point, we will simply ignore writes through that pointer. Second, if we stop symbolic execution of a loop after *max_pass* iterations, we may retain facts about pointers that are inaccurate, causing writes through that pointer to update the wrong variables. Neither of these issues has been a problem in practice.

Pistachio only derives a fact if the theorem prover can show that the write is within bounds, and otherwise kills all existing facts about the array. Note that even though a buffer overflow may modify other memory, we do not kill other facts, which is unsound but helps reduce false positives. Also, since all C data is represented as byte arrays, type casts are implicitly handled as well,

CheckSingleRule(R, *Out*, S)

Input:
R — The rule to check
Out — The initial facts at each program point
S — The statements at which to start checking R
Output: A tuple $(T, status)$, plus updates the fact set *Out*
T — The statements where rule verification ended
status — Whether R verified or not

```
1  Let R be of the form (P_H, H) ⇒ (P_C, C, G)
2  for s ∈ S do
3      Out(s) ← Out(s) ∪ H ∪ P_H(s)
4  end for
5  Let T be the set of statements that are the first matches to P_C on all paths from statements in S
6  If there is a path from S on which no statement matches P_C, return (T, rule not satisfied)
7  FactDerivation(Out, S, T)
8  if ∀t ∈ T, Theorem-prover((Out(t) ∧ P_C(t)) ⇒ C) = yes then
9      /* R is satisfied */
10     for s' ∈ T do
11         Remove from Out(s') facts involving ghost variables modified in G
12         Out(s') ← Out(s') ∪ G
13     end for
14     Remove from Out all the facts involving pattern variables
15     return (T, rule satisfied)
16 else
17     /* R is not satisfied */
18     return (T, rule not satisfied)
19 end if
```

Fig. 6. Algorithm for checking a single rule

as long we can determine at analysis time the allocation size for each type, which Pistachio could always do in our experiments. In addition, in order to reduce false positives Pistachio assumes that variables are initialized with 0's, even when locally scoped.

Pistachio assumes that code is single-threaded, and is not generally sound in the presence of multi-threading, since it does not simulate thread interleavings. However, it can still be used quite effectively on multi-threaded programs if we assume that any threads are completely independent and do not share any communication-related state, as is the case for our SSH benchmarks.

In the next sections, we illustrate the use of *FactDerivation* during the process of checking the alternating bit protocol from Fig. 2.

3.1 Checking a Single Rule

Given the *FactDerivation* algorithm, we can now present our algorithm for checking that the code obeys a single rule R of the form $(P_H, H) \Rightarrow (P_C, C, G)$. Assume that we are given a set of statements S that match P_H. Then to check R, we need to simulate the program forward from the statements in S using *FactDerivation*. We check that we can reach statements matching P_C along all paths and that the conclusion C holds at those statements. Fig. 6 gives our formal algorithm *CheckSingleRule* for carrying out this process.

The input to *CheckSingleRule* is a rule R, an initial set of facts *Out*, and a set of starting statements S. For all statements in S, on line 3 we add to their facts the assumptions H and any facts derived from pattern matching S against P_H; we denote this latter set of facts by $P_H(s)$. On line 5 we search forward along all program paths until we first find the conclusion pattern P_C. If there is some path from S along which we cannot find a match to P_C, we exit on line 6, reporting that the rule is not satisfied. Otherwise, on line 7 we perform symbolic execution using *FactDerivation* to update *Out*. On line 8, we use the theorem prover to check whether the conclusion C holds at the statements that match P_C. If they do then the rule is satisfied, and lines 11–12 update $Out(s')$ with facts for ghost variables. We also remove any facts about pattern variables (*in* and *out* in our examples) from *Out* (line 14), return T, and indicate the rule was satisfied. Otherwise on line 18, we return T but report that the rule was not satisfied.

Note that *pattern variables* (such as *in* and *out* in our examples) that occur in a rule R have a scope limited to the program paths followed by Pistachio to verify R. Once the verification is complete (line 14), all facts about pattern variables are removed. On the other hand, *ghost variables* (such as n in our examples) have a global scope. This means that when Pistachio encounters a ghost variable *gvar*, facts about it will persist between rule firings and can only be "killed" by the successful verification of another rule that modifies *gvar*. Pistachio also kills facts about C variables when they go out of scope.

We illustrate using *CheckSingleRule* to check rule (1) from Fig. 2 on the CFG in Fig. 3(a). The first block in Fig. 3(b) lists the steps taken by the algorithm. We will discuss the remainder of this figure in Section 3.2.

In rule (1), the hypothesis pattern P_H is the start of the program, and the set of assumptions H is empty. The conclusion C of this rule is $out[0..3] = 1$, where *out* matches the second argument passed to a call to `send()`. Thus to satisfy this rule, we need to show that $out[0..3] = 1$ at statement 2 in Fig. 1. We begin by adding H and $P_H(0)$, which in this case are empty, to $Out(0)$, the set of facts at the beginning of the program, which is also empty. We trace the program from this point forward using *FactDerivation*. In particular, $Out(1) = Out(2) = \{val = 1\}$. At statement 2 we match the call to `send()` against P_C, and thus we also have fact $\&val = out$. Then we ask the theorem prover to show $Out(2) \wedge \{\&val = out\} \Rightarrow C$. In this case the proof succeeds, and so the rule is satisfied, and we set ghost variable n to 1.

3.2 Checking a Set of Rules

Finally, we develop our algorithm for checking a set of rules. Consider again

CheckRuleSet($\mathcal{R}$)

Input: $\mathcal{R}$ — the set of rules to check

```
 1 Let start be the initial statement of the program
 2 Out(start) = ∅
 3 ∀s ≠ start, set Out(s) = ⊤, where ⊤ is the set of all possible facts
 4 W ← ∅
 5 for R ∈ R do
 6     if R has the form (ε, ∅) ⇒ (P_C, C, G) then
 7         W ← W ∪ {(R, Out, {start})}
 8         R ← R − {R}
 9     end if
10 end for
11 while W not empty do
12     (R, Out, S) ← dequeue(W)
13     visit(R) ← visit(R) + 1
14     if visit(R) > max_pass then
15         continue
16     end if
17     Let R be of the form (P_H, H) ⇒ (P_C, C, G)
18     Let T be the set of statements that are the first matches to P_H on all paths from statements in S
19     FactDerivation(Out, S, T)
20     for s ∈ T do
21         if Theorem-prover(Out(s) ∧ H ⇒ false) = yes then
22             T ← T − {s}
23         end if
24     end for
25     if T ≠ ∅ then
26         (U, status) ← CheckSingleRule(R, Out, T)
27         if status = rule satisfied then
28             for R ∈ R do
29                 W ← W ∪ {(R, Out, U)}
30             end for
31         else
32             Report that R fails to check
33         end if
34     end if
35 end while
```

Fig. 7. Algorithm for checking a set of rules

the rules in Fig. 2. Notice that rules (2) and (3) both depend on n, which is set in the conclusion of rules (1) and (2). Thus we need to check whether rules (2) or (3) are triggered on any program path after we update n, and if they are, then we need to check whether they are satisfied. Since rule (2) depends on itself, we in fact need to iterate.

Fig. 7 gives our algorithm for checking a set of rules. The input is a set of rules $\mathcal{R}$ that need to be checked. On lines 2–3, we create an initial *Out*, which is empty for the initial statement in the program, and $\top$, the set of all facts, for every other program point. The main body of the algorithm maintains a worklist W containing tuples (R, Out, S), where R is a rule to be checked starting at statements in S with the set of facts *Out*. Line 4 initializes W to be empty, and then for each rule R that has $(\varepsilon, \emptyset)$ as the hypothesis (and so matches the beginning of the program and has no hypotheses), line 7 adds $(R, Out, \{start\})$ to W. Notice that this means a copy of *Out* is stored in the worklist for each initial rule. Rules with the empty hypothesis are also removed from $\mathcal{R}$, since they will not be checked again.

The main body of the algorithm (lines 11–35) iteratively removes an element (R, Out, S) from the worklist and checks the rule. Note that because rule dependencies may be cyclic, we may visit a rule multiple times, and as in Fig. 6 we terminate iteration once we have visited a rule *max_pass* times (lines 13–16). On line 18 we trace forward from S to find all statements T that match P_H. Then on line 19 we perform fact derivation to update *Out*. This updates only the local copy of *Out* we removed from the worklist, and not any other fact sets. The set T now contains the statements where we should start checking the rule R. First, however, lines 20—24 remove any statements from T for which the rule hypotheses and the current state are provably inconsistent (i.e., imply *false*), meaning that the rule cannot fire. Assuming that T is non-empty after this process, line 26 invokes *CheckSingleRule*, which updates *Out* and returns a new set of statements U and a status. If the rule was satisfied, then lines 28–30 add (R, Out, U) to the worklist for all rules R, so that future iterations of the loop will check any new rules that may fire starting from U. If the rule was not satisfied, then line 32 reports a rule violation.

We illustrate the algorithm by tracing through the remainder of Fig. 3(b), which describes the execution of *CheckRuleSet* on our example program. Observe that rule (1) has no assumptions and matches the beginning of the program, hence it is initially added to W. The main loop of the algorithm removes $((1), Out, \{start\})$ from the worklist, and then calls *CheckSingleRule*. As described in Section 3.1, we satisfy rule (1) at statement 2, and we set ghost variable n to 1. Let Out' be the current fact set. The we add $((2), Out', \{2\})$ and $((3), Out', \{2\})$ to the worklist. (We do not add rule (1) to the worklist, because it matched the start of the program.) Thus either rule (2) or rule (3) might fire next.

Checking rule (3). Suppose we next remove $((3), Out', \{2\})$ from the worklist, meaning we check rule (3) starting at statement 2, in state Out', which has the ghost variable n set to 1. We perform symbolic execution forward until we find a statement that matches the hypothesis pattern of rule (3), which is statement 4. Now we check that rule (3) should fire, which it does, because the current state does not contradict the hypothesis $in[0..3] \neq n$ (in fact, it does not say anything about *in*). Thus we add the hypothesis to our set of facts and continue forward. When we reach statement 5, the theorem prover shows that the false branch will be taken, so we only continue along that one branch—which is important, because if we followed the other branch we would not be able to prove the conclusion. Taking the false branch, we fall through to statement 7, and the theorem prover concludes that rule (3) holds. Since the rule is satisfied, we add back rules (2) and (3) to the worklist to be checked starting at the bottom of the loop; we will come back to this below.

Checking rule (2). Next suppose we remove $((2), Out', \{2\})$ from the worklist, meaning that we need to now check rule (2) starting where we left off at

statement 2. Out' has $n = 1$ and $val = 1$ at statement 2. We continue forward, match the hypothesis of rule (2) at statement 4, and then this time at the conditional we conclude that the true branch is taken, hence *val* becomes 3 at statement 7. Now the conclusion of the rule cannot be proven, so we issue a warning that the protocol was violated at this statement.

Finding a fixpoint. Suppose for illustration purposes that rule (2) had checked successfully, i.e., in statement 6, `val` was only incremented by one. Then at statement 7 we would have shown the conclusion and set ghost variable $n := out[0..3]$. Then we would add $((2), Out'', \{7\})$ and $((3), Out'', \{7\})$ to the worklist, where Out'' is the state after rule (2) was checked. When we remove these elements from the worklist, we will recheck those rules for the loop body with the new value of n, and then continue to repeat this process, leading to an infinite loop, cut off after *max_pass* times. A similar effect happens after checking rule (3) on the body of the loop.

However, notice that after one iteration, *Out* stabilizes. During the first check of rule (2), we would have that $Out(2) = \{n = val, val = 1\}$, which is the input fact set for statement 4. After rule (2) succeeds at statement 7, we set n to *val*, and hence $Out(7) = \{n = val, val = 2\}$. Thus when we revisit statement 4, we intersect $Out(2)$ and $Out(7)$, yielding the set $\{n = val\}$. This is, in fact, a loop invariant, and subsequent checks of rules (2) and (3) will wind up re-checking the same rules in the same state, producing no new results.

Pistachio includes two features relating to fixpoints that are not shown in our formal algorithms, because they would obscure their structure. First, Pistachio stops iteration when it detects a fixpoint, i.e., when we either have already simulated forward from some point with the same set of facts, or when we have already checked a rule at the same statement starting with the same set of facts. Second, in the cases where Pistachio does not find a fixpoint, rather than intersecting fact sets on back edges, Pistachio performs loop unrolling. Loops are unrolled until we either reach a fixpoint, unroll a total of *max_pass* iterations, or provably exit the loop before *max_pass* iterations are reached. We found that Pistachio was very often able to find a fixpoint and stop iterating well before reaching *max_pass* in our experiments (Section 4.5).

4 Implementation and Experiments

Pistachio is implemented in approximately 6,000 lines of OCaml. We use CIL [14] to parse C programs and the Darwin [15] automated theorem prover (for a discussion on the choice of theorem prover, see Appendix B). Darwin is a sound, fully-automated theorem prover for first-order clausal logic. Darwin can return either "yes," "no," or "maybe" for each theorem. Pistachio con-

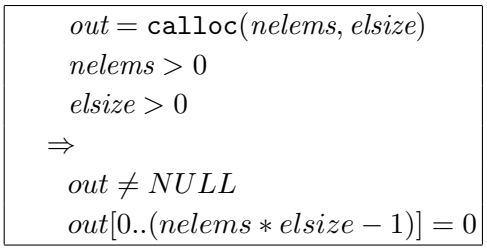


Fig. 8. Example rule for library function `calloc`

servatively reports a rule failure if the theorem prover cannot prove that the facts derived starting from the hypothesis imply the conclusion.

To analyze realistic programs, Pistachio needs to model system libraries. Rather than analyze library source code directly, which is not always available, Pistachio allows the user to provide a rule-based specification for library functions. For example, Fig. 8 gives a library rule that models calls to `calloc`. The pattern in the hypothesis of the rule works just like a regular rule, matching a function call, here with *nelems* and *elsize* bound to the parameters and *out* bound to its return value. Library rules only fire if the predicates in the hypothesis are implied by the current set of facts, and if they are, the conclusion facts are added directly after the call. (Note the difference with a regular rule, where the hypothesis facts cannot be contradictory but are otherwise assumed, the conclusion contains a pattern, and the conclusion facts must be proven when the pattern matches.) For example, if Pistachio sees a call `block = calloc(n, sz)` at statement s, Pistachio asks the theorem prover to show

$$In(s) \wedge (nelems = \texttt{n}) \wedge (elsize = \texttt{sz}) \wedge (out = \texttt{block}) \Rightarrow (nelems > 0) \wedge (elsize > 0)$$

If the answer is *yes*, then Pistachio adds the facts $\{\texttt{block} \neq NULL, \texttt{block}[0..(\texttt{n} * \texttt{sz} - 1)] = 0\}$ to $Out(s)$. Otherwise the library rule is ignored. Notice that our rule for `calloc` assumes that infinite memory is available.

For our experiments we wrote a total of 117 library rule specifications for 35 I/O and memory allocation routines. Library specifications can also be used for user-defined functions, in which case they override the function definition. In our SSH2 experiments, we used 18 specifications for cryptographic functions, rather than analyze their code. (Our specifications treat cryptographic functions as if they perform no encryption, since that is not our concern in these experiments.) There are some library functions we cannot fully model in our framework, e.g., `geterrno()`, which potentially depends on any other library call. Pistachio assumes there are no side-effects from calls to library functions without specifications, and Pistachio assumes nothing about the return value of the call. For example, Pistachio assumes that conditionals based on the result of `geterrno()` may take either branch, which can reduce precision.

Category	Example rule(s)	Description
Message structure and data transfer (SSH2: 35, RCP: 19)	`recv`(*in_sock*, *in*, _) *in.msgid* = *SSH_MSG_USERAUTH_REQUEST* *in.authtype* = *"keyboard-interactive"* ⇒ `send`(*out_sock*, *out*, _) *in_sock* = *out_sock* *out.msgid* = *SSH_MSG_USERAUTH_INFO_REQUEST* *len*(*out.prompt*) > 0	In keyboard interactive authentication mode, the prompt field(s) [the server sends to the interactive client] MUST NOT be empty.
Compatibility (SSH2: 11, RCP: 7)	`recv`(*sock*, *in*, _) *in*[*len*(*in*) − 2] = *CR* *in*[*len*(*in*) − 1] = *LF* *connected*[*sock*] = 0 ⇒ `send`(*sock*, *out*, _) *out*[*len*(*out*) − 2] = *CR* *out*[*len*(*out*) − 1] = *LF* *connected*[*sock*] := 1	In compatibility mode, after receiving the client identification string, the server MUST NOT send any additional messages before its identification string. [Note: The message with the identity string is distinguished by ending in a CR/LF combination]
Functionality (SSH2: 24, RCP: 17)	`recv`(_, *in*, _) *in*[0] = *SSH_MSG_USERAUTH_REQUEST* *isOpen*[*in*[1..4]] = 1 *in*[21..25] = *"none"* ⇒ `send`(_, *out*, _) *out*[0] = *SSH_MSG_USERAUTH_FAILURE*	It is STRONGLY RECOMMENDED that the "none" authentication method not be supported.
Protocol logic (SSH2: 26, RCP: 15)	`recv`(*in_sock*, *in*, _) *in*[0] = *SSH_MSG_GLOBAL_REQUEST* *in*[1..14] = *"tcpip-forward"* *in*[15] = 1 *in*[(*len*(*in*) − 4)..(*len*(*in*) − 1)] = 0 ⇒ `send`(*out_sock*, *out*, _) *in_sock* = *out_sock* *out*[0] = *SSH_MSG_REQUEST_SUCCESS*	The server MUST respond to a TCP/IP forwarding request in which the *wantreply* flag [byte 15] is set to 1 and the *port* [last 4 bytes] is set to 0 with a *SSH_MSG_-REQUEST_SUCCESS* containing the forwarding port.
	`recv`(*in_sock*, *in*, _) *in*[0] = *SSH_MSG_GLOBAL_REQUEST* *in*[15] = 1 ⇒ `send`(*out_sock*, *out*, _) *in_sock* = *out_sock*	The server MUST respond to all global requests with the *wantreply* flag [byte 15] set to 1.

Fig. 9. Rule categorization and counts, and example rules for SSH2

4.1 Core Rule Sets

We evaluated Pistachio by analyzing the LSH [16] and OpenSSH [17] implementations of SSH2 and the RCP implementation from Cygwin's inetutils package. We chose these systems because of their extensive bug databases and the number of different versions available.

We created rule-based specifications by following the process described in Section 2.2. Our (partial) specification for SSH2 totaled 96 rules, and the one for RCP totaled 58 rules. The rules for SSH2 covered the authentication, connec-

tion and transport parts of the protocol, but none of its extensions (such as key exchange protocols). Because the rules describe particular protocol details and are interdependent, it is difficult to correlate individual rules with general correctness or security properties. In Fig. 9, we give a rough classification of the rules in our specifications, list how many rules in each specification fall into each category, and show example rules from SSH2. These example rules are taken directly from our experiments—the only changes are that we have reformatted them in more readable notation, rather than in the concrete grammar used in our implementation, and we have used `send` and `recv` rather than the actual function names.

There are four rule categories, based on functional behavior:

- **Message structure and data transfer** includes rules that relate to the format, structure, and restrictions on messages in the protocols. The example rule requires that any prompt sent to keyboard interactive clients in SSH2 not be the empty string. RCP rules in this category specify properties such as payload length or structure of messages about negotiated transmission parameters.
- **Compatibility** includes rules about backwards compatibility with earlier protocol versions. The example rule places a requirement on sending an identification string when SSH2 is in compatibility mode with SSH1.
- **Functionality** includes rules about what abilities should or should not be supported. The example rule requires that an SSH2 implementation not allow the "none" authentication method. RCP rules in this category check that the required authentication and encryption methods are available.
- **Protocol logic** contains the most complex rules, and rules in this category were the most time-consuming to develop. These rules require that the proper response is sent for each received message. The first example SSH2 rule requires that the server provide an adequate response to TCP/IP forwarding requests with a value of 0 for *port*. The second SSH2 rule requires that the server replies to all global requests that have the *wantreply* flag set. RCP rules in this category specify how the implementation should respond to expected and out-of-phase data and control messages, or the way transmission parameters such as payload size and frequency are negotiated.

Based on our experience developing rules for SSH and RCP, we believe that the process does not require any specialized knowledge other than familiarity with the rule-based language grammar and the specification document. It took the authors less than 7 hours to develop each of our SSH2 and RCP specifications.

Of course, we did not get the rules entirely right the first time we wrote them down. We debugged our set of rules for SSH by identifying a set of 20 bugs they should catch and then running Pistachio on the corresponding implementations. We selected our set of bugs from four LSH versions (0.1.9,

0.5, 0.7.3, and 1.2.5) that were not used in our evaluation, and we chose bugs that were local to those versions—i.e., they did not appear in or persist to the next version of the protocol. As a result of this process, we made modifications to 9 of the 87 initial SSH2 rules. The process took approximately 1.5 hours out of the total 7 hours to develop and debug the specification. We repeated the same process for RCP using 14 bugs from versions 0.7.1 and 0.9 (which were also not used in our evaluation). As a result, we modified 6 of the 51 RCP rules, and the debugging process took roughly 45 minutes. The value of 75 for *max_pass* was determined on the same set of versions used to debug the rules.

We debugged Pistachio itself on simplified versions of the sliding window protocol. Since sliding window is the basis of many modern protocols (including TCP/IP), variations of it (often simplified to one-bit message payloads) have been thoroughly studied in the literature [18], both from the point of view of correctness and that of the implementation. We debugged Pistachio on the Stop-And-Wait, Go-Back-n, and Selective-Repeat variations of the protocol from five implementations freely available on the web. When we performed our experiments on LSH and RCP, we also found an fixed a few other bugs in our implementation.

Generally the rules are slightly more complex than is shown in Fig. 9. On average, rules in the SSH2 specification include 11 hypothesis and 5 conclusion facts, and rules in the RCP specification include 9 hypothesis and 4 conclusion facts. Originally, we started with a rule set derived directly from specification documents, which was able to catch many but not all bugs, and then we added some additional rules (a little over 10% more per specification) to look for some specific known bugs. These additional rules produced about 20% of the warnings from Pistachio (Section 4.5). Generally, the rules written from the specification document tend to catch missing functionality and message format related problems, but are less likely to catch errors relating to compatibility problems or unsafe use of library functions. The process of extending the initial set of rules proved fairly easy, since specific information about the targeted bugs was available from the respective bug databases.

After developing our SSH2 rules, including the extra 10% mentioned above, we conducted an initial experiment in which we applied them to LSH [10]. Subsequently, we have ported the rules to also work on OpenSSH, and in this paper we are using the new set of rules for both LSH and OpenSSH. Since the rules were based on the specification document and did not depend on specific programming constructs, 94 of our original 96 SSH2 rules could be applied to OpenSSH with no changes. The remaining two rules needed to change because of differences in how the LSH and OpenSSH authors interpreted the protocol description. In particular, the SSH2 connection protocol specification [6] states:

LSH

Version	0.1.3	0.2.1	0.2.9	0.9.1	1.0.1	1.1.1	1.2.1	1.3.1	1.4.1	1.5.1	1.5.5	2.0.1
LoC	8,745	8,954	9,145	9,267	9,221	10,431	10,493	10,221	12,585	12,599	12,754	13,689
Time (s)	24	26	25	27	27	28	29	30	31	33	38	40
Warnings	118	123	110	97	91	93	91	79	78	70	33	25
Bugs	86	79	66	61	77	77	80	80	50	39	31	12
Bugs in db	91	83	69	65	81	80	82	82	51	40	31	13
False pos	40	54	42	43	22	26	24	14	33	35	9	12
False neg	5	4	3	4	4	3	2	2	1	1	0	1

OpenSSH

Version	1.0 p1	1.0 p2	1.2 p5	1.2 p7	1.2.1 p18	1.2.1 p20	1.2.1 p23	1.2.1 p24	1.2.1 p27	1.2.2	1.2.3	2.0.1
LoC	10,546	12,467	13,612	15,432	19,834	20,312	21,400	21,514	21,543	24,315	24,532	24,761
Time (s)	37	40	39	42	43	44	44	48	48	49	50	55
Warnings	51	48	42	38	36	39	31	30	31	21	11	9
Bugs	39	37	26	24	23	24	23	19	18	10	5	1
Bugs in db	43	39	29	27	26	28	25	22	20	11	5	1
False pos	11	15	13	10	8	11	18	14	11	6	4	5
False neg	4	2	3	3	3	4	2	3	2	1	0	1

RCP

Version	0.5.4	0.6.4	0.8.4	1.1.4	1.2.3	1.3.2
LoC	4,501	4,723	4,813	4,865	4,891	5,026
Time (s)	17	19	20	19	23	26
Warnings	72	78	58	70	48	39
Bugs	48	44	46	49	28	22
Bugs in db	51	46	47	51	28	23
False pos	30	30	17	25	27	19
False neg	3	2	1	2	0	1

Fig. 10. Pistachio results with core rule sets

X11 connection forwarding should stop when the session channel is closed. However, already opened forwardings should not be automatically closed when the session channel is closed.

Our set of rules targeting LSH assumed that existing forwardings are not in fact closed, which holds in that implementation. OpenSSH, however, does close existing forwardings when the session channel is closed, if they have been idle for a predefined period of time. Adjusting for this difference accounts for the two rule changes. The experimental results in the following sections used the revised SSH2 rules for both LSH and OpenSSH.

4.2 Results for Core Rule Sets

We started with initial specifications for the SSH2 protocols (87 rules) and the RCP protocol (51 rules), with "core rules" based only on specification

documents. In Section 4.5 we discuss extending these rules to target particular bugs. Using these specifications, we ran our tool against selected versions of LSH, OpenSSH, and RCP. We used the same specification for all versions of the code, and we ran Pistachio with *max_pass* set to 75.

Fig. 10 presents our analysis results. For each program version, we list the lines of code, omitting comments and blank lines, and the running time of Pistachio. We only include code analyzed by Pistachio and involved in the rules. We also list Pistachio's running time, in seconds. The running times were measured on a Pentium IV 3Ghz machine with 1GB of RAM running SuSE Linux 9.3. The times were measured as an average of five runs, and include the time for checking the core rules plus the additional rules discussed in Section 4.5. In all cases the running times were under one minute.

The remaining rows measure Pistachio's effectiveness. We list the number of warnings generated by Pistachio, the number of actual bugs those warnings correspond to, the number of bugs in the project's bug database for that version, and the number of false positives—warnings that do not correspond to bugs in the code—and false negatives—bugs in the database we did not find. Note that there is not always a one-to-one correspondence between bugs and Pistachio warnings. When counting bugs in the database, we count only reports for components covered by our specifications. For instance, we only include bugs in SSH2's authentication, transport and connection protocol code, but not any code that is part of the key exchange protocol. We also did not include reports that appeared to be feature requests or other issues.

We found that most of the warnings reported by Pistachio correspond to bugs that were in the database. For the core rule sets, the average false negative rates were quite low (4% for LSH, 9% for OpenSSH, and 3% for RCP), and the average false positive rates were fairly low (35% for LSH, 38% for OpenSSH, and 42% for RCP). We were pleasantly surprised by the very low false negative rate, especially since Pistachio is neither sound nor complete. We also noted that Pistachio is able to handle pointer arithmetic and aliasing very well, as a significant percentage of warnings and corresponding bugs are related to data access. These results suggest that Pistachio can be an effective tool for finding bugs in network protocol implementations.

Section 4.4 discusses some of the bugs we found in detail. Pistachio discovered two apparently new bugs in LSH version 2.0.1. The first is a buffer overflow in buffers reserved for passing environment variables in the SSH Transport protocol. The second involves an incorrectly formed authentication failure message when using PKI. We have not confirmed these bugs; we sent an email to the LSH mailing list with a report, but it appears that the project has not been maintained recently, and we did not receive a response.

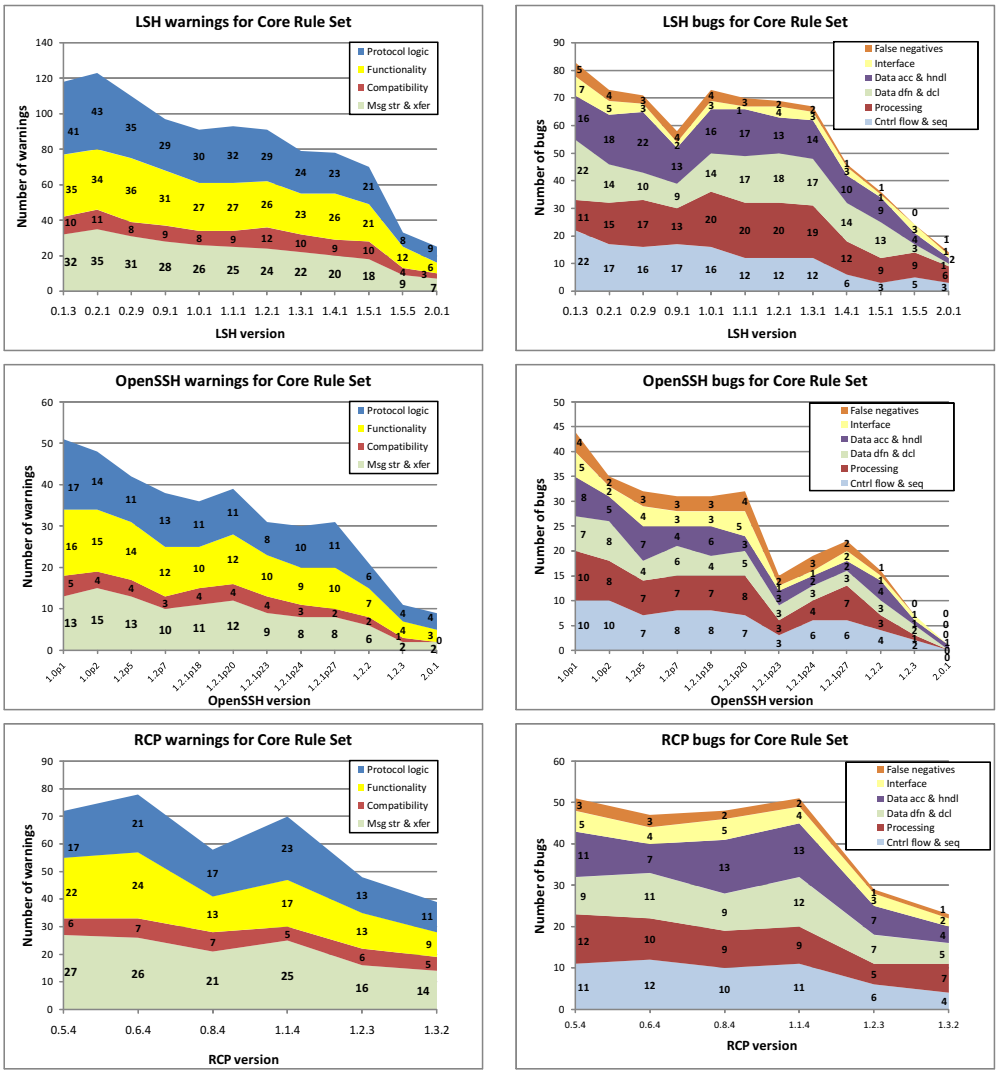


Fig. 11. Warnings by Pistachio and bugs detected for LSH, OpenSSH and RCP

4.3 Breakdown of warnings and bugs

Next, we wanted to get a finer view of how Pistachio behaves, to understand two things: First, are there particular categories of rules that tend to result in more warnings than other categories? And second, does Pistachio tend to find certain kinds of bugs more than other kinds?

To answer the first question, we broke down the warning counts by the rule categorization in Fig. 9. The results are shown in the left half of Fig. 11. For each version, listed along the bottom, we show the number of warnings due to violations of protocol logic, functionality, compatibility, and message structure and transfer rules. From these graphs, we can see that the category with the fewest warnings overall is *compatibility*, but there are no clear standouts among the other rule categories. There is one interesting difference between RCP

and the SSH2: For the simpler RCP protocol, the number of protocol logic warnings tends to be lower than the number of warnings related to message structure and transfer, while this situation is often reversed for the SSH2 protocol implementations. Overall, our results show that each of the different kinds of rules have a good chance of catching programmer mistakes; or, put another way, that programmers make mistakes that span all rule categories. These graphs also show quite clearly that the total number of warnings goes down over time, which makes sense, since we hope that software improves as it matures.

While measuring how often rule categories are violated gives us some sense of how the warnings are distributed by rule, it does not always tell us about the underlying defects in the program. For example, a rule failure could be caused by missing parentheses in the code, by a mis-constructed if-then-else, or by a multitude of other mistakes. Thus, to answer our second question, which defects cause rule violations, we classified bugs found by Pistachio into a subset of Beizer's bug taxonomy [9], one popular classification system.

Beizer's taxonomy is large and complex, containing eight main categories, three of which, *Structural*, *Data*, and *Implementation*, contain elements related to code (as opposed to specifications, requirements, etc.). Within these top-level categories, we selected second-level categories that related to the kinds of problems Pistachio found. We divided defects into the following five categories:

- **Control flow and sequencing** contains bugs related to branch and loop conditions and statements. Examples include missing or misplaced *break* statements, wrong initial values of terminal conditions for loops, and incorrect conditions for exceptionally exiting loops.
- **Processing** refers to errors in expressions and computation in general. Examples include misplaced parentheses in an arithmetic expression, insufficient or excessive precision, incorrect bitwise masks, and cutting off the beginning or end or a string.
- **Data definition and declaration** refers to bugs related to data declaration and initialization. Examples include forgetting to initialize a variable, incorrect data types (e.g., short instead of long), global/local inconsistencies or conflicts, and incorrect allocation type (e.g., static when it should have been dynamic).
- **Data access and handling** refers primarily to indirect memory accesses. Examples include writing through an invalid pointer, incorrect type casts, and object boundary errors.
- **Interface** refers to errors in the passing of parameters to user-defined or library methods. Examples include passing fewer variables to `sprintf` than type specifiers, duplicate or spurious method calls, and accessing the wrong return parameter.

Deciding which category a bug falls into is an inexact process. For example, suppose in a loop we have moved a pointer past the end of an array and we attempt to write through that pointer. This is clearly a buffer overflow, but in the taxonomy, it could be categorized as a *control flow and sequencing* bug, since the termination condition on the loop was incorrect; or as a *data definition and declaration* bug, if the problem was that the array was declared with too small a size; or as a *data access and handling* bug, since we attempted to write through an invalid pointer. When categorizing bugs, we always tried to place the bug in the last category in which it fits, in the order the categories are listed above. Thus in our example situation, the bug would fall under *data access and handling*.

The right side of Fig. 11 shows the results of this classification; we also include false negatives for comparison. These graphs show that no one kind of defect dominates overall, though there are relatively few *interface* bugs. For LSH and OpenSSH, *processing* bugs are often the most numerous, where RCP tends to have the most bugs in *data access and handling* and *control flow and sequencing*, though these trends are subtle and may not be significant. Beizer reports [9] that the five defect categories we chose account for roughly 61% of total bugs in software, according to data collected between 1982 and 1988 from US defense, aerospace, and telecommunication companies. Interestingly, in Beizer's data, all five categories are roughly equivalent in their contribution, whereas as we just observed, we found relatively few interface bugs. We suspect that this is related to the scale of the code in our experiments. The protocol implementations are comprised of only a few modules, which were likely written by only one or a few people, minimizing the chance of a miscommunication about an interface. Excepting interface bugs, our results suggest that problems in network protocol implementations are due to the usual range of programming mistakes.

4.4 Discussion

Our results show that Pistachio can find a wide variety of bugs in network protocol implementations. In this section, we discuss the process a user needs to following in order to understand the output of Pistachio, i.e., to determine whether a warning corresponds to a potential bug. We also describe several sample bugs we found in LSH, discuss Pistachio's false positives and negatives, and examine the security implications of the bugs we found.

Understanding Pistachio Warnings We found that the best way to determine whether a rule violation corresponds to a bug is to trace the fact derivation process in reverse, using logs produced by Pistachio that give *Out*

for each analyzed statement. We start from a rule failure at a conclusion, and then examine the preceding statements to see whether they directly contradict those facts. If so, then we have found what we think is a bug, and we look in the bug database to determine whether we are correct. If the preceding statements do not directly contradict the facts, we continue tracing backward, using *Out* to help understand the results, until we either find a contradiction or reach the hypothesis, in which case we have found a false positive. If the conclusion pattern is not found on all program paths, we use the same process to determine why those paths were not pruned during fact derivation.

As a concrete example of this process, consider the second rule in Fig. 9. This rule is derived from section 5 of the SSH2 Transport Protocol specification [6], and Pistachio reported that this rule was violated for LSH implementation 0.2.9. In particular, Pistachio could not prove that $out[len(out)-2] = CR$ and $out[len(out)-1] = LF$ at the rule conclusion. Fig. 12(a) shows the code (slightly simplified for readability) where the bug was reported. Statement 24 matches the conclusion pattern (here *fmsgsend* is a wrapper around *send*), and so that is where we begin. Statement 24 has two predecessors in the control-flow graph, one for each branch of the conditional statement 8. Looking through the log, we determined that the required facts to show the conclusion were in *Out*(19) but not in *Out*(13). We examined statement 13, and observed that if it is executed (i.e., if the conditional statement 8 takes the true branch), then line 24 will send a disconnect message, which is incorrect. Thus we backtracked to statement 8 and determined that *protomsg.proto_ver<1* could not be proved either true or false based on *In*(8), which was correctly derived from the hypothesis (asserted in *Out*(3)). Thus we determined that we found a bug, in which the implementation could send an *SSH_DISCONNECT* message for clients with versions below 1.0, although the protocol specifies that the server must send the identification first, independently of the other party's version. We then confirmed this bug against the LSH bug database. This bug falls under *control flow and sequencing*, since the *if* branch in the code should be removed.

While it is non-trivial to determine whether the reports issued by Pistachio correspond to bugs, we found it was generally straightforward to follow the process described above. Usually the number of facts we were trying to trace was small (at most 2-3), and the number of preceding statements was also small (rarely larger than 2). In the vast majority of the cases, it took on the order of minutes to trace through a Pistachio warning, though in some cases it took up to an hour, often due to insufficiently specified library functions that produced many paths. In general, the effort required to understand a Pistachio warning is directly proportional to the complexity of the code making up a communication round.

```
1  // Code extracted from a function
2  // handling connection initialization
3  fmsgrecv(clisock, inmsg, SSH2_MSG_SIZE);
4  if (!parse_message(MSGTYPE_PROTOVER,
5                     inmsg, len(inmsg), &protomsg))
6    return;
7  ...
8  if (protomsg.proto_ver < 1) {
9    payload.msgid = SSH_DISCONNECT;
10   payload.reason =
11     SSH_DISCONNECT_PROTOCOL_ERROR;
12 ...
13   sz = pack_message(MSGTYPE_DISCONNECT,
14                     payload, outmsg,
15                     SSH2_MSG_SIZE);
16 } else {
17   sprintf(outstr, "\%.1f\%c\%s\%c\%c",
18           2, SP, SRV_COMMENTS, CR, LF);
19   sz = pack_message(MSGTYPE_PROTOVER,
20                     outstr, outmsg,
21                     SSH2_MSG_SIZE);
22 ...
23 }
24 fmsgsend(clisock, outmsg, sz);
```

(a) Compatibility Rule Violation

```
1  fmsgrecv(clisock, inmsg, SSH2_MSG_SIZE);
2  if (!parse_message(MSGTYPE_USERAUTHREQ,
3                     inmsg, len(inmsg), &authreq))
4    return;
5  ...
6  if (authreq.method==USERAUTH_PKI) {
7  ...
8  } else if (authreq.method==USERAUTH_PASSWD) {
9  ...
10 } else {
11 ...
12 }
13 sz = pack_message(MSGTYPE_REQSUCCESS,
14                   payload, outmsg,
15                   SSH2_MSG_SIZE);
16 fmsgsend(clisock, outmsg, sz);
```

(b) Functionality Rule Violation

```
1  char laddr[17]; int lport;
2  ...
3  fmsgrecv(clisock, inmsg, SSH2_MSG_SIZE);
4  if (!parse_message(MSGTYPE_GLOBALREQ,
5                     inmsg, len(inmsg), &globreq))
6    return;
7  ...
8  if (globreq.msgtype==
9        MSGSUBTYPE_TCPIPFORWARD) {
10   strcpy(laddr, getstrfield(globreq.payload, 0));
11   lport = getuint32field(globreq.payload, 1);
12 ...
13   if (!create_forwarding(clisock, laddr, lport))
14     return debug_error();
15   if ((globreq.wantreply==1) &&
16       (lport == 0)) {
17     payload.msgid =
18       SSH_REQUEST_SUCCESS;
19     payload.reason = lport;
20     sz =
21       pack_message(MSGTYPE_REQSUCCESS,
22                    payload, outmsg,
23                    SSH2_MSG_SIZE);
24     fmsgsend(clisock, outmsg,sz);
25   }
26 }
```

(c) Buffer Overflow

```
1  fmsgrecv(clisock, inmsg,
2    SSH2_MSG_CHANNEL_REQUEST);
3  if (!parse_message(MSGTYPE_CHREQ,
4      inmsg, len(inmsg),&chreq))
5    return;
6  ...
7  if (chreq.msgtype==MSGSUBTYPE_SHELL) {
8  ...
9    /* fmod was previously set to ''rw'' */
10   if (!(clish = popen(make_clishell(clisock), fmod)))
11     return debug_error()};
```

(d) Library Call Error

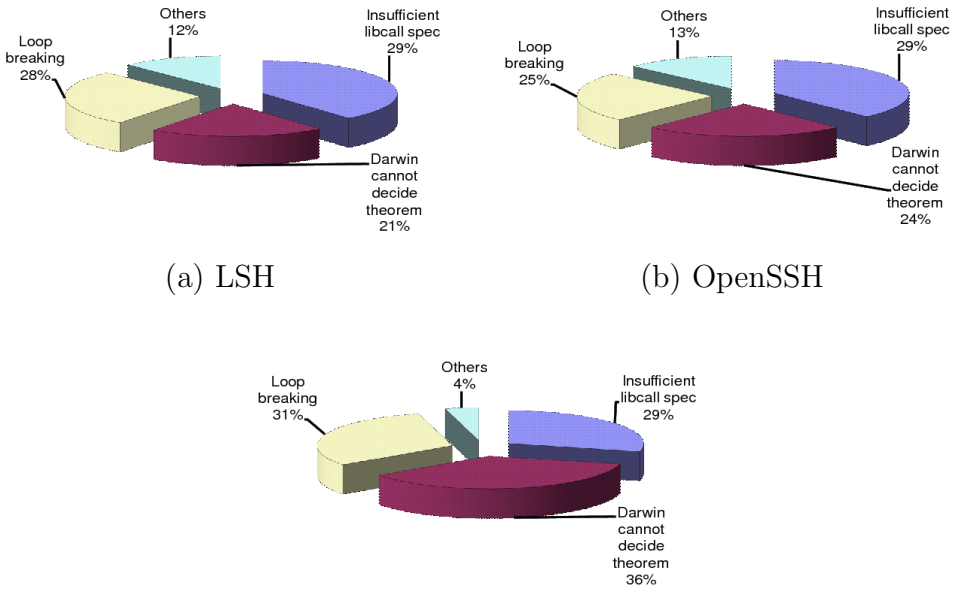
Fig. 12. Sample bugs in LSH

More Sample Bugs Found by Pistachio Fig. 12(b) shows a violation of the example functionality rule in Fig. 9. This code is from LSH version 0.1.3. In this case, a message is received at statement 1, and Pistachio assumes the rule hypotheses, which indicate that the message is a user authorization request. Then a success message is always sent in statement 16. However,

the rule specifies that the "none" authentication method must result in a failure response. Tracing back through the code, we discovered that the *else* statement on line 10 allows the "none" method to succeed. This statement should have checked for the "hostbased" authentication method, and indeed this corresponds to a bug in the LSH bug database. This is also a *control flow and sequencing* bug, as there is a missing branch case for "hostbased" authentication.

Fig. 12(c) shows a buffer overflow detected by Pistachio in LSH version 0.9.1. Buffer overflows are detected indirectly during rule checking. Recall that when Pistachio sees a write to an array that it cannot prove is in-bounds, it kills facts about the array. Thus sometimes when we investigated why a conclusion was not provable, we discovered it was due to an out-of-bounds write corresponding to a buffer overflow. When we ran Pistachio on the code in Fig. 12(c), we found a violation of the first protocol logic rule in Fig. 9, as follows. At statement 3, Pistachio assumes the hypothesis of this rule, including that the *wantreply* flag (corresponding to *in*[15]) is set, and that the message is for TCP forwarding. Under these assumptions, Pistachio reasons that the true branch of statement 8 is taken. But then line 10 performs a *strcpy* into laddr, which is a fixed-sized locally-allocated array. The function *getstrfield()* (not shown) extracts a string out of the received message, but that string may be more than 17 bytes. Thus at the call to *strcpy*, there may be a write outside the bounds of laddr, and so we kill facts about laddr. Then at statement 13, we call *create_forwarding()*, which expects laddr to be null-terminated—and since we do not know whether there is a null byte within laddr, Pistachio determines that *create_forwarding()* might return false, causing us to return from this code without executing the call to *fmsgsend* in statement 24. This bug falls under the *data declaration and definition* category, since the statically allocated *laddr* should in this case be dynamic based on the length of the message payload (or alternatively, the payload should be truncated).

In this case, if Pistachio had been able to reason at statement 10 that laddr was null-terminated, then it would not have issued a warning. Although the return statement 14 might seem to be reachable in that case, looking inside of *create_forwarding()*, we find that can only occur if LSH runs out of ports, and our model for library functions assumes that this never happens. (Even if an ill-formed address is passed to *create_forwarding()*, it still creates the forwarding for 0.0.0.0.) On the other hand, if *create_forwarding()* had relied on the length of laddr, rather than it being null-terminated, then Pistachio would not have reported a warning here—even though there still would be a buffer overflow in that case. Thus the ability to detect buffer overflows in Pistachio is somewhat fragile, and it is a side effect of the analysis that they can be detected at all. Buffer overflows that do not result in rule violations, or that occur in code we do not analyze, will go unnoticed.



(a) LSH (b) OpenSSH

(c) RCP

Fig. 13. Causes for false positives

Finally, Fig. 12(d) shows a bug in LSH 0.9.1 found as a violation of the last rule in Fig. 9. The problem in this code had to do with a library function, and this error, like a buffer overflow, was found as a side effect of rule checking. In this code, we matched the hypothesis of the rule at statement 1, and then determined that the branch on statement 7 may be taken. Thus one possible path leads to the call to *popen* in statement 10. In this case, our model of *popen* requires that the second argument must be either "r" or "w," or the call to *popen* yields an undefined result. Since before statement 10 fmod was set to "rw," Pistachio assumes that *popen* may return any value, including null, and thus statement 11 may be executed and return without sending a reply message, thus violating the rule conclusion. Note that our model of *popen* always succeeds if valid arguments are passed, and thus if fmod were "r" or "w" a rule violation would not have been reported. This bug falls under the *interface* category, since it involves incorrect function arguments.

False Positives and Negatives Fig. 13 breaks down the causes of false positives found in LSH, OpenSSH, and RCP, averaged over all versions. The main cause of false positives is insufficient specification of library calls. This is primarily due to the fact that library functions sometimes rely on external factors such as system state (e.g., whether *getenv()* returns NULL or not depends on which environment variables are defined) that cannot be fully modeled using our rule-based semantics. For such functions, only partial specifications can be devised. The remaining false positives are due to limitations of the theorem prover and to loop breaking, where we halt iteration of our algorithm after *max_pass* times.

Besides false positives, Pistachio also has false negatives, as measured against the LSH, OpenSSH, and RCP bug databases. From our experience, these are generally caused by either assumptions made when modeling library calls, or by the fact that the rule sets are not complete. As an example of the first case, we generally make the simplifying assumption that on `open()` calls, there are sufficient file handles available. This caused a false negative for LSH version 0.1.3, where a misplaced `open()` call within a loop lead to the exhaustion of available handles for certain SSH global requests.

Security Implications As can be seen from the previous discussion, many of the bugs found by Pistachio have obvious security implications. In general, categorizing a bug as security-relevant is difficult, because bugs that initially appear benign may introduce security vulnerabilities, and the definition of what is and is not a vulnerability depends on particular circumstances. We looked through the bug databases to determine which bugs found by Pistachio are either clearly security-related by their nature or were documented as security-related. On average, we classified 30% of the true positive warnings (warnings that correspond to bugs) as security-related for LSH, 28% for OpenSSH, and 23% for RCP.

Of the security-related bugs, 52% are buffer overflows (all of which we assume are security-related), 21% have to do with access control (ensuring the user has sufficient privileges before performing operations), and 18% are compatibility problems. The last category does not directly violate security, but does impede the use of a secure protocol. The remaining security-related bugs did not fall into any broader categories.

Our classification of bugs as security-related has some uncertainty, because the bug databases might incorrectly categorize some non-exploitable bugs as security holes. Conversely, some bugs that are not documented as being security-related might be exploitable by a sufficiently clever attacker. In general, any bug in a network protocol implementation is undesirable.

4.5 Results for Extended Rule Sets

When bugs are found in code, it is good software engineering practice to write regression tests to catch the bug if it reappears. We hypothesized that in a similar way, Pistachio could be integrated into the development lifecycle by writing "regression rules" that target known bugs.

We conducted a second set of exploratory experiments in which we extended our core rule-based specifications to target bugs that they missed according

LSH

Version	0.1.3	0.2.1	0.2.9	0.9.1	1.0.1	1.1.1	1.2.1	1.3.1	1.4.1	1.5.1	1.5.5	2.0.1
Additional warnings	**12**	**20**	**13**	**14**	**15**	**16**	**16**	**21**	**17**	**8**	**8**	**3**
Msg str & xfer	3	6	5	3	4	3	4	5	4	2	2	1
Compatibility	1	2	2	1	1	1	1	2	2	1	.	.
Functionality	4	7	3	4	4	5	5	7	5	3	4	1
Protocol logic	4	5	3	6	6	7	6	7	6	2	2	1
Additional bugs	**2**	**2**	**2**	**4**	**1**	**3**	**1**	**2**	**1**	.	.	.
Cntrl flow & seq	1	1	.	.	.	1	.	1	.	.	.	.
Processing	.	.	1	1	1	.	.	.	1	.	.	.
Data dfn & dcl	1	1	.	1	.	2	1	.	.	.	.	.
Data acc & hndl	.	.	.	2	.	.	.	.	.	.	.	.
Interface	.	.	1	.	.	.	.	1	.	.	.	.
Bugs in database	**91**	**83**	**69**	**65**	**81**	**80**	**82**	**82**	**51**	**40**	**31**	**13**
Additional false pos	**7**	**8**	**6**	**8**	**7**	**5**	**6**	**5**	**7**	.	**3**	.
Remaining false neg	**3**	**2**	**1**	.	**3**	.	**1**	.	.	**1**	.	**1**

OpenSSH

Version	1.0 p1	1.0 p2	1.2 p5	1.2 p7	1.2.1 p18	1.2.1 p20	1.2.1 p23	1.2.1 p24	1.2.1 p27	1.2.2	1.2.3	2.0.1
Additional warnings	**9**	**13**	**10**	**3**	**3**	**5**	**5**	**3**	**3**	**4**	**5**	**2**
Msg str & xfer	2	3	2	1	1	1	1	1	1	1	1	.
Compatibility	1	1	2	.	.	.	1	.	.	.	1	.
Functionality	3	4	4	1	2	2	1	1	.	2	2	1
Protocol logic	3	5	2	1	.	2	2	1	2	1	1	1
Additional bugs	**2**	**1**	**1**	**3**	**1**	**4**	**1**	**2**	**1**	**1**	.	.
Cntrl flow & seq	1	1	.	.	.	1	.	1	.	.	.	.
Processing	.	.	.	1	.	1	.	.	1	.	.	.
Data dfn & dcl	.	.	.	.	1	.	.	.	.	1	.	.
Data acc & hndl	1	.	1	2	.	1	1	1	.	.	.	.
Interface	.	.	.	.	.	1	.	.	.	.	.	.
Bugs in database	**43**	**39**	**29**	**27**	**26**	**28**	**25**	**22**	**20**	**11**	**5**	**1**
Additional false pos	**6**	**8**	**6**	**3**	**4**	**1**	**3**	**6**	**3**	**2**	**2**	**2**
Remaining false neg	**2**	**1**	**2**	.	**2**	.	**1**	**1**	**1**	.	.	.

RCP

Version	0.5.4	0.6.4	0.8.4	1.1.4	1.2.3	1.3.2
Additional warnings	**12**	**11**	**9**	**7**	**8**	**6**
Msg str & xfer	3	2	2	2	2	1
Compatibility	1	.	1	1	.	.
Functionality	4	4	2	2	2	2
Protocol logic	4	5	4	2	4	3
Additional bugs	**3**	**1**	.	**2**	.	**1**
Cntrl flow & seq	1	.	.	1	.	.
Processing	1	.	.	.	.	.
Data dfn & dcl	.	.	.	.	.	.
Data acc and hndl	1	1	.	1	.	1
Interface	.	.	.	.	.	.
Bugs in database	**51**	**46**	**47**	**51**	**28**	**23**
Additional false pos	**6**	**5**	**5**	**3**	**2**	**4**
Remaining false neg	.	**1**	**1**	.	.	.

Fig. 14. Pistachio results with extended rule sets

to the bug databases. Our goal was to determine whether new rules could catch these bugs. We added 9 new rules to our SSH2 specification, and 7 new rules to our RCP specification, or roughly 10% more rules overall. The rules we needed to add were typically for features that were strongly recommended but not required by the specification, because it turned out that violations of these recommendations were considered errors. One example is the recommendation that a proper disconnect message be sent by the SSH server when authentication fails.

Fig. 14 shows the number of additional warnings, bugs, and false positives found by Pistachio with the extra rules. We also break down the warnings and bugs as in Section 4.3; we do not graph the data since the numbers are small. In this figure, zero values are entered as a dot.

The warnings and bugs span the gamut of our categorization, though there are almost no interface bugs. Interestingly, the new SSH2 rules were determined by only looking at the LSH bug database, and yet they also found additional bugs in OpenSSH. Overall, the additional rules account for under 20% of the total number of warnings generated by Pistachio. Of the new warnings reported by Pistachio, approximately 17% had security implications according to our classification from Section 4.4, mostly related to access control issues and buffer overflows. Fig. 14 also lists the number of false negatives that remain even after enriching the specifications. Roughly half of the remaining false negatives are due to terminating iteration after *max_pass* times, and the other half are due to aspects of the protocol our new rules still did not cover.

For the extended rules, we also measured how often we are able to compute a symbolic fixpoint for loops during our analysis. Recall that if we stop iteration of our algorithm after *max_pass* times then we could introduce unsoundness, which accounts for approximately 28% of the false positives, as shown in Fig. 13. We found that when *max_pass* is set to 75, we find a fixpoint before reaching *max_pass* in 250 out of 271 cases for LSH, in 204 out of 252 for OpenSSH, and in 153 out of 164 cases for RCP. This suggests that our symbolic fixpoint computation is effective in practice.

5 Related Work

Understanding the safety and robustness of network protocols is recognized as an important research area, and the last decade has witnessed an emergence of many techniques for verifying protocols.

We are aware of only a few systems that, like Pistachio, directly check source code rather than abstract models of protocols. CMC [1] and VeriSoft [19] both

model check C source code by running the code dynamically on hardware, and both have been used to check communication protocols. These systems execute the code in a simulated environment in which the model checker controls the execution and interleaving of processes, each of which represents a communication node. As the code runs, the model checker looks for invalid states that violate user-specified assertions, which are similar to the rules in Pistachio. CMC has been successfully used to check an implementation of the AODV routing protocol [1] and the Linux TCP/IP protocol [20].

There are two main drawbacks to these approaches. First, they potentially suffer from the standard state space explosion problem of model checking, because the number of program executions and interleavings is extremely large. This is typical when model checking is used for data dependent properties, and both CMC and VeriSoft use various techniques to limit their search. Second, these tools find errors only if they actually occur during execution, which depends on the number of simulated processes and on what search algorithm is used. Pistachio makes different tradeoffs. Because we start from a set of rules describing the protocol, we need only perform symbolic execution on a single instance of the protocol rather than simulating multiple communication nodes, which improves performance. The set of rules can be refined over time to find known bugs and make sure that they do not appear again. We search for errors by program source path rather than directly in the dynamic execution space, which means that in the best case we are able to use symbolic information in the dataflow facts to compute fixpoints for loops, though in the worst case we unsafely cut off our search after *max_pass* iterations. Pistachio is also very fast, making it easy to use during the development process. On the other hand, Pistachio's rules cannot enforce the general kinds of temporal properties that model checking can. We believe that ultimately the CMC and VeriSoft approach and the Pistachio approach are complementary, and both provide increased assurance of the safety of a protocol implementation.

Other researchers have proposed static analysis systems that have been applied to protocol source code. MAGIC [21] extracts a finite model from a C program using various abstraction techniques and then verifies the model against the specification of the program. MAGIC has been successfully used to check an implementation of the SSL protocol. The SPIN [2] model checker has been used to trace errors in data communication protocols, concurrent algorithms, and operating systems. It uses a high level language to specify system descriptions but also provides direct support for the use of embedded C code as part of model specifications. However, due to the state space explosion problem, neither SPIN nor MAGIC perform well when verifying data-driven properties of protocols, whereas Pistachio's rules are designed to include data modeling. Feamster and Balakrishnan [4] define a high-level model of the BGP routing protocol by abstracting its configuration. They use this model to build *rcc*, a static analysis tool that detects faults in the router configuration. Naumovich

et al. [5] propose the FLAVERS tools, which uses dataflow analysis techniques to verify Ada pseudocode for communication protocols. Alur and Wang [22] formulate the problem of verifying a protocol implementation with respect to its standardized documentation as refinement checking. Implementation and specification models are manually extracted from the code and the RFC document and are compared against each other using reachability analysis. The method has been successfully applied to two popular network protocols, PPP and DHCP. In the context of cryptographic protocols, Bhargavan et al.[23] propose fs2pv, a prototype tool to extract a verifiable formal model from implementation code of protocols written in F#. The formal model is then verified against realistic threat models using the ProVerif [24] automatic verification tool, which we describe in the next paragraph. It is unclear whether fs2pv can verify non-cryptographic properties of protocols, or can be adapted to C, which many protocols are implemented in.

Many systems have been developed for verifying properties of abstract protocol specifications. In these systems the specification is written in a specialized language that usually hides some implementation details. These methods can perform powerful reasoning about protocols, and indeed one of the assumptions behind Pistachio is that the protocols we are checking code against are already well-understood, perhaps using such techniques. The main difficulty of checking abstract protocols is translating RFCs and other standards documents into the formalisms and in picking the right level of abstraction. $Mur\varphi$ is a system for checking protocols in which abstract rules can be extracted from actual C code [25]. Similarly to Pistachio, the user has to specify a list of variables and functions relevant to the properties to be checked as well as define the correctness properties of the protocol model. The main differences between our approach and the $Mur\varphi$ system lie in how the rules are interpreted: in $Mur\varphi$ the rules are an abstraction of the system and are derived automatically, whereas in Pistachio rules specify the actual properties to be checked in the code. Uppaal [3] models systems (including network protocols) as timed automata, in which transitions between states are guarded by temporal conditions. This type of automata is very useful in checking security protocols that use time challenges and has been used extensively in the literature to that extent [26,27]. In [27], Uppaal is used to model check the TLS handshake protocol. CTL model checking can also be used to check network protocols. In [28], an extension of CTL is used to model AODV. Butler et al. [29,30] rely on the MSR formalization language [31,32] to provide precise specifications for the Kerberos authentication protocol. The specifications are in the form of a finite set of transitions that define all possible executions from a fixed initial state. The properties of the protocol are expressed as formal theorems and verified using theorem proving techniques. The NRL Protocol Analyzer [33] models cryptographic protocols [34] as a set of communicating state machines, each of which with multiple local state variables. To determine if a state is insecure, the analyzer works backwards from the state until all

the possible transitions have been either explored or discarded. In ProVerif [24], cryptographic protocols are abstracted by a set of Prolog clauses that correspond to the execution of the protocol itself and to the computational abilities and knowledge of an attacker. A novel resolution-based algorithm similar to the Prolog solving algorithm is then used to prove secrecy properties. Similarly to Pistachio, the correct behavior of the protocol is represented through a set of rules. However, in ProVerif, the rules fully model the protocol and are sufficient to determine whether the protocol is faulty. On the other hand, Pistachio works directly with the implementation, and its rules specify only partial correctness properties to be checked using symbolic execution and theorem proving.

Recently there has been significant research effort on developing static analysis tools for finding bugs in software. We list a few examples: SLAM [35] and BLAST [36] are model checking systems that have been used to find errors in device drivers. MOPS [37] uses model checking to check for security property violations, such as TOCTTOU bugs and improper usage of *setuid*. Metal [38] uses data flow analysis and has been used to find many errors in operating system code. ESP [39] uses data flow analysis and symbolic execution, and has been used to check sequences of I/O operations in gcc. CQual [40–42] uses type qualifier inference to find a variety of bugs in C programs, including security vulnerabilities in the Linux kernel.

All of these tools have been effective in practice, and allow the user to specify the property to be checked. However, they are restricted to checking finite state properties, i.e., safety properties that can be modeled by associating finite automata with program locations (local variables and the heap); the automaton takes transitions when certain actions happen in the program, and the tool issues a warning if an automaton enters an error state. Some examples of finite state properties are: forbidding double acquires and releases of the same lock (a bug if locks are non-reentrant) [41,35]; forbidding double frees of the same memory location [38]; and making sure files are open in the right mode before being read or written [39]. It is unclear whether these tools can effectively check the kinds of rich, data-dependent rules used by Pistachio.

Dynamic analysis can also be used to trace program executions, although we have not seen this technique used to check correctness of implementations. Gopalakrishna et al. [43] define an Inlined Automaton Model (IAM) that is flow- and context-sensitive and can be derived from source code using static analysis. The model is then used for online monitoring for intrusion detection.

Another approach to finding bugs in network protocols is online testing. Protocol fuzzers [44] are popular tools that look for vulnerabilities by feeding unexpected and possibly invalid data to a protocol stack. Because fuzzers can find hard-to-anticipate bugs, they can detect vulnerabilities that a Pistachio

user might not think to write a rule for. On the other hand, the inherent randomness of fuzzers makes them hard to predict, and sometimes finding even a single bug with fuzzing may take a long time. Pistachio quickly checks for many different bugs based on a specification, and its determinism makes it easier to integrate in the software development process.

Our specification rules are similar to precondition/postcondition semantics usually found in software specification systems or design-by-contract systems like JML [45]. Similar constructs in other verification systems also include BLAST's event specifications [36].

6 Conclusion

We have defined a rule-based method for the specification of network protocols which closely mimics protocol descriptions in RFC or similar documents. We have then shown how static analysis techniques can be employed in checking protocol implementations against the rule-based specification and provided details about our experimental prototype, Pistachio. Our experimental results show that Pistachio is very fast and is able to detect a number of security-related errors in implementations of the SSH2 and RCP protocols, while maintaining low rates of false positives and negatives.

Acknowledgments

This research was supported in part by NSF CCF-0346982 and CCF-0430118. We thank Mike Hicks, David Wagner, Nick Petroni, R. Sekar, and the anonymous reviewers for their helpful comments.

References

[1] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, D. L. Dill, CMC: A Pragmatic Approach to Model Checking Real Code, in: Symposium on Operating Systems Design and Implementation (OSDI), 2002, pp. 75–88.

[2] G. J. Holzmann, The Model Checker SPIN, Software Engineering 23 (5) (1997) 279–295.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL - a tool suite for automatic verification of real-time systems, in: Hybrid Systems, 1995, pp. 232–243.

[4] N. Feamster, H. Balakrishnan, Detecting BGP Configuration Faults with Static Analysis, in: Symposium on Networked Systems Design and Implementation (NSDI), 2005.

[5] G. N. Naumovich, L. A. Clarke, L. J. Osterweil, Verification of Communication Protocols Using Data Flow Analysis, in: Symposium on Foundations of Software Engineering, 1996, pp. 93–105.

[6] T. Ylonen, C. Lonvick, The Secure Shell (SSH) Connection Protocol, RFC 4254 (Proposed Standard) (Jan. 2006).
URL `http://www.ietf.org/rfc/rfc4254.txt`

[7] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Symposium on Principles of Programming Languages (POPL), 1977, pp. 238–252.

[8] N. Möller, lshd leaks fd:s to user shells, `http://lists.lysator.liu.se/pipermail/lsh-bugs/2006q1/000467.html` (Jan. 2006).

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

[10] O. Udrea, C. Lumezanu, J. S. Foster, Rule-based static analysis of network protocol implementations, in: Usenix Security Symposium, 2006, pp. 193–208.

[11] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, A Note on Reliable Full-duplex Transmission over Half-duplex Links, Communications of the ACM 12 (5) (1969) 260–261.

[12] J. Postel, J. Reynolds, Instructions to RFC Authors, RFC 2223 (Informational) (Oct. 1997).
URL `http://www.ietf.org/rfc/rfc2223.txt`

[13] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1988.

[14] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, in: International Conference on Compiler Construction, 2002, pp. 213–228.

[15] P. Baumgartner, A. Fuchs, C. Tinelli, Darwin: A Theorem Prover for the Model Evolution Calculus, in: IJCAR Workshop on Empirically Successful First Order Reasoning, 2004.

[16] A GNU implementation of the Secure Shell protocols, `http://www.lysator.liu.se/~nisse/lsh/`.

[17] OpenSSH, `http://www.openssh.com`.

[18] D. E. Comer, Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architectures, Fourth Edition, Prentice Hall PTR, 2000.

[19] P. Godefroid, Model Checking for Programming Languages Using Verisoft, in: Symposium on Principles of Programming Languages (POPL), 1997, pp. 174–186.

[20] M. Musuvathi, D. R. Engler, Model checking large network protocol implementations, in: Symposium on Networked Systems Design and Implementation (NSDI), 2004, pp. 155–168.

[21] S. Chaki, E. Clarke, A. Groce, S. Jha, H. Veith, Modular Verification of Software Components in C, in: International Conference on Software Engineering, 2003, pp. 385–395.

[22] R. Alur, B.-Y. Wang, Verifying network protocol implementations by symbolic refinement checking, in: International Conference on Computer-Aided Verification, 2001, pp. 169–181.

[23] K. Barghavan, C. Fournet, A. D. Gordon, S. Tse, Verified interoperable implementations of security protocols, in: Computer Security Foundations Workshop, 2006.

[24] B. Blanchet, An efficient cryptographic protocol verifier based on prolog rules, in: Computer Security Foundations Workshop, 2001.

[25] D. Lie, A. Chou, D. Engler, D. L. Dill, A Simple Method for Extracting Models for Protocol Code, in: International Symposium on Computer Architecture, 2001, pp. 192–203.

[26] S. Yang, J. S. Baras, Modeling Vulnerabilities of Ad Hoc Routing Protocols, in: ACM Workshop on Security of Ad Hoc and Sensor Networks, 2003, pp. 12–20.

[27] G. Diaz, F. Cuartero, V. Valero, F. Pelayo, Automatic Verification of the TLS Handshake Protocol, in: ACM Symposium on Applied Computing, 2004, pp. 789–794.

[28] R. Corin, S. Etalle, P. H. Hartel, A. Mader, Timed Model Checking of Security Protocols, in: ACM Workshop on Formal Methods in Security Engineering, 2004, pp. 23–32.

[29] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, C. Walstad, Formal analysis of kerberos 5, Theoretical Computer Science 367 (1) (2006) 57–87.

[30] I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, C. Walstad, Breaking and fixing public-key kerberos, in: Asian Computing Science Conference, 2006.

[31] I. Cervesato, Typed msr: Syntax and examples, in: MMM-ACNS '01: International Workshop on Information Assurance in Computer Networks, 2001.

[32] N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov, Multiset rewriting and the complexity of bounded security protocols, Journal of Computer Security 12 (2) (2004) 247–311.

[33] C. Meadows, The NRL protocol analyzer: An overview, Journal of Logic Programming 26 (2) (1996) 113–131.

[34] C. Meadows, Analysis of the internet key exchange protocol using the nrl protocol analyzer, in: IEEE Security and Privacy, 1999.

[35] T. Ball, S. K. Rajamani, The SLAM Project: Debugging System Software via Static Analysis, in: Symposium on Principles of Programming Languages (POPL), 2002, pp. 1–3.

[36] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software Verification with BLAST, Lecture Notes in Computer Science 2648 (2003) 235–239.

[37] H. Chen, D. Wagner, MOPS: An Infrastructure for Examining Security Properties of Software, in: ACM Conference on Computer and Communications Security, 2002, pp. 235–244.

[38] D. Engler, D. Y. Chen, S. Hallem, A. Chou, B. Chelf, Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, in: ACM Symposium on Operating Systems Principles (SOSP), 2001, pp. 57–72.

[39] M. Das, S. Lerner, M. Seigle, ESP: Path-Sensitive Program Verification in Polynomial Time, in: ACM Conference on Programming Languages Design and Implementation (PLDI), 2002, pp. 57–68.

[40] J. S. Foster, R. Johnson, J. Kodumal, A. Aiken, Flow-Insensitive Type Qualifiers, ACM Trans. Prog. Lang. Syst. 28 (6) (2006) 1035–1087.

[41] J. S. Foster, T. Terauchi, A. Aiken, Flow-Sensitive Type Qualifiers, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, 2002, pp. 1–12.

[42] R. Johnson, D. Wagner, Finding User/Kernel Bugs With Type Inference, in: Proceedings of the 13th Usenix Security Symposium, San Diego, CA, 2004.

[43] R. Gopalakrishna, E. H. Spafford, J. Vitek, Efficient intrusion detection using automaton inlining, in: IEEE Symposium on Security and Privacy, 2005, pp. 18–31.

[44] P. Oehlert, Violating assumptions with fuzzing, IEEE Security & Privacy Magazine 3 (2005) 58–62.

[45] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, B. Jacobs, JML: notations and tools supporting detailed design in Java, in: International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Companion, 2000, pp. 105–106.

[46] D. Blei, C. Harrelson, R. Jhala, R. Majumdar, D. C. Necula, S. P. Rahul, W. Weimer, D. Weitz, Vampyre version 1.0, `http://www.cs.ucla.edu/~rupak/Vampyre` (2000).

[47] Hol 4, `hol.sourceforge.net`.

$$
\begin{aligned}
R &::= \texttt{(RULE}\ id\ Hyp\ \texttt{=>}\ Con\texttt{)} \\
Hyp &::= \varepsilon \mid Pat\ Fact^* \\
Con &::= Pat\ Fact^*\ Ghost^* \\
\\
Pat &::= \texttt{(RET}\ Base\texttt{)} \mid \texttt{(CALL}\ id\texttt{(}Base^*\texttt{))} \\
Base &::= id \mid \_ \\
Fact &::= \texttt{(}Comp\ Expr\ Expr\texttt{)} \\
Ghost &::= \texttt{(SET}\ Var\ Expr\texttt{)} \\
\\
Var &::= id \mid Var\texttt{[}Idx\texttt{]} \mid Var.id \\
Idx &::= Expr \mid Expr\texttt{..}Expr \\
Expr &::= Var \mid \langle integer \rangle \mid \langle double \rangle \mid \langle string \rangle \\
&\quad \mid \texttt{(}Op\ Expr\ Expr\texttt{)} \mid \texttt{(}UOp\ Expr\texttt{)} \\
Op &::= Comp \mid \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \\
UOp &::= \texttt{len} \\
Comp &::= \texttt{=} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=}
\end{aligned}
$$

Fig. A.1. Pistachio rule-based specification grammar

A Rule Grammar

Fig. A.1 gives the grammar for rules R in Pistachio. This grammar is slightly generalized from the actual implementation for clarity. Rules are written in S-expression notation to make them easier for Pistachio to parse. Each rule R is named with an identifier *id* and contains a hypothesis and conclusion. Each hypothesis or conclusion contains exactly one pattern, which may either be the empty pattern, match a return statement, or match a function call. The expressions used within a pattern, described by non-terminal *Base*, are restricted to identifiers or the wildcard pattern `_`. Hypotheses and conclusions also contain a set of facts, which are assertions about simple expressions with no side effects. Expressions can refer to identifiers, and may access array elements of fields. Within expressions *Expr* we allow basic operations such as addition and subtraction. We also allow `len`, which returns the length of an array (or produces an error if Pistachio does not know the length of the array). Conclusions may also set ghost variables, which we write with an S-expression beginning with `SET` rather than the infix `:=` notation used in the main text of the paper.

B Choosing a theorem prover

While we chose Darwin as Pistachio's automatic theorem prover, we also experimented with using Vampyre [46] and HOL [47]. Vampyre is a sound theo-

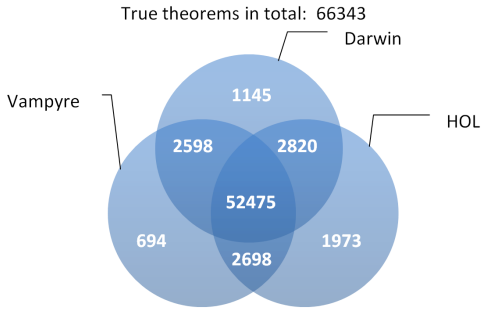


Fig. B.1. Theorems proved by Darwin, Vampyre and HOL

rem prover based on the Nelson-Oppen architecture. Although it was written to produce explicit proofs of verification conditions for proof carrying code applications, it can also be used as a general purpose theorem prover. As the name indicates, HOL is a higher order logic [1] theorem prover. Since Darwin and Vampyre are written in OCaml, as is Pistachio, it is easy to call them directly to represent theorems and obtain proofs. And since HOL uses ML (of which OCaml is a variant) to represent its theorems, proofs, and even proof strategies, and we were easily able to transform the theorems needed by Pistachio into the format required by HOL.

We compared all three theorem provers by running Pistachio on the benchmarks described in Section 4. Across all runs, we counted the total number of theorems Pistachio asked that were true, and the number of those true theorems each prover was able to prove. (Counting these was a non-trivial task, but was not as hard as it seems—we assumed that any theorem provable by one of the theorem provers was true, and thus only had to manually check the remaining theorems, which were plentiful but tended to be small.)

Figure B.1 gives a Venn diagram of the results. Overall, the three provers had very similar accuracy, and there were some true theorems that none of the provers were able to show. We can see that the large majority of theorems succeeded for all three provers (52,475 out of a total of 66,343 theorems). Note that this is only a rough indication of utility, since we did not evaluate how the differences affected false positive and negative rates. Despite the very similar accuracy, Darwin had the best running time—an average of 0.0024 seconds per theorem, compared to 0.113 seconds for Vampyre and 0.679 seconds for HOL. This, along with the ease of use of its OCaml interface compared to Vampyre and HOL, are the reasons we chose Darwin as the theorem prover for Pistachio.

[1] A version of predicate calculus with types and variables ranging over functions and predicates.