8.0 CONCORD - VERIFYING MEMORY SAFETY

In this part of the project, we set out to explore how to move very precise static analysis and verification techniques from the realm of specialized research tools to an approach that can feasibly adopted by programmers in the real world. The critical issue is the lack of precision that such analyses inevitably encounter when analyzing the programs that occur in practice, no matter how advanced the automated techniques are. This lack of precision causes either 1) unacceptable numbers of false positive alarms or 2) the use of unsound techniques that may leave errors uncovered.

Our hypothesis we set out to test is that, in existing programs, only a small percentage of the code is responsible for this lack of precision. If this hypothesis is true, it opens up the possibility that we are much closer to a practical program analysis for verifying important security (and potentially other properties) than it currently appears.

For this, we developed a focused prototype version of a highly precise verifier based on the Compass tool [92] we call Concord. While our tool is only a prototype, we were able to test and validate our initial hypothesis and concretise the key challenges currently limiting the impact of program verification as well as formulate initial solutions to these challenges.

We have validated on real code that loss of analysis precision is generally caused by a small subset of the code. But more specifically, this insight leads to two technical challenges:

1. Where the precision loss starts is often far from where a spurious error is reported and very hard to identify by hand.

2. Once the piece of code too difficult to analyze is identified, non-expert programmers need to specify a correct alternate description or implementation.

In our project, we propose using *logical abduction* to identify the parts of a code base that introduce analysis imprecisions. In order for non-expert programmers to annotate, replace and/or describe the behavior of code segments that are beyond automatic analysis capabilities, we propose a technique we call *property programming* where programmers rewrite the small sections of code that are too hard to analyze. Where this is not possible (e.g. stubbing libraries, or very critical code segments), a small set of intuitive annotation primitives are embedded into regular program constructs (such as loops) to allow non-expert programmers to easily and succinctly express difficult constraints and invariants.

8.1 Logical Abduction

8.1.1 General Introduction

The fundamental ingredient of automated logical reasoning is *deduction*, which allows deriving valid conclusions from a given set of premises. For example, consider the following set of facts:

$$\begin{array}{ll} (1) & \forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x)) \\ (2) & \forall x.\ ((\text{duck}(x) \vee \text{goose(x)}) \Rightarrow \text{waddle}(x)) \\ (3) & \text{duck}(\text{donald}) \end{array}$$

Based on these premises, logical deduction allows us to reach the conclusion:

$$\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})$$

This form of forward deductive reasoning forms the basis of all SAT and SMT solvers as well as first-order theorem provers and verification tools used today.

A complementary form of logical reasoning to deduction is *abduction*, as introduced by Charles Sanders Peirce [93]. Specifically, abduction is a form of backward logical reasoning, which allows inferring likely premises from a given conclusion. Going back to our earlier example, suppose we know premises (1) and (2), and assume that we have observed that the formula $\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})$ is true. Here, since the given premises do not imply the desired conclusion, we would like to find an explanatory hypothesis ψ such that the following deduction is valid:

$$\frac{\begin{array}{c} \forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x)) \\ \forall x.\ ((\text{duck}(x) \vee \text{goose(x)}) \Rightarrow \text{waddle}(x)) \\ \psi \end{array}}{\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})}$$

The problem of finding a logical formula ψ for which the above deduction is valid is known as *abductive inference*. For our example, many solutions are possible, including the following:

$$\begin{array}{ll} \psi_1: & \text{duck}(\text{donald}) \wedge \neg\text{quack}(\text{donald}) \\ \psi_2: & \text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald}) \\ \psi_3: & \text{goose}(\text{donald}) \wedge \text{quack}(\text{donald}) \\ \psi_4 & \text{duck}(\text{donald}) \end{array}$$

While all of these solutions make the deduction valid, some of these solutions are more desirable than others. For example, ψ_1 contradicts known facts and is therefore a useless solution. On the other hand, ψ_2 simply restates the desired conclusion, and despite making the deduction valid, gets us no closer to explaining the observation. Finally, ψ_3 and ψ_4 neither contradict the premises nor restate the conclusion, but, intuitively, we prefer ψ_4 over ψ_3 because it makes fewer assumptions.

At a technical level, given premises Γ and desired conclusion ϕ, abduction is the problem of finding an explanatory hypothesis ψ such that:

$$\begin{array}{ll} (1) & \Gamma \wedge \psi \models \phi \\ (2) & \Gamma \wedge \psi \not\models \text{false} \end{array}$$

Here, the first condition states that ψ, together with known premises Γ, entails the desired conclusion ϕ, and the second condition stipulates that ψ is consistent with known premises. As illustrated by the previous example, there are many solutions to a given abductive inference problem, but the most desirable solutions are those that are as simple and as general as possible.

Recently, abductive inference has found many useful applications in verification, including inference of missing function preconditions [94, 95], diagnosis of error reports produced by verification tools [96], and for computing under-approximations [97]. Furthermore, abductive inference has also been used for inferring specifications of library functions [98] and for automatically synthesizing circular compositional proofs of program correctness [99].

In the context of the Concord project, our goal is to utilize abduction to identify the *smallest* and *most general* annotations required to verify a program. Assume that everything a static analysis could automatically learn about a program is encoded in constraint ψ and we are trying to prove a property encoded in constraint ϕ. By definition, if our tool reports a potential error, it must be that

$$\psi \not\models \phi$$

Therefore finding a smallest root cause of the error reported can be directly mapped into logical abduction

$$\begin{array}{ll} (1) & \Gamma \wedge \psi \models \phi \\ (2) & \Gamma \wedge \psi \not\models \text{false} \end{array}$$

where ψ is a smallest piece of information that, if true and annotated by the user, will make the original condition provable. Of course the smallest such fact may not actually be true, therefore we need to generate a sequence of abductive solutions of increasing difficulty until the programmer using the tool confirms one of them by placing the right annotations.

8.1.2 Algorithm for Performing Abductive Inference

In this section, we describe the algorithm used in for performing abductive inference at a high level. First, let us observe that the entailment $\Gamma \wedge \psi \models \phi$ can be rewritten as $\psi \models \Gamma \Rightarrow \phi$. Furthermore, in addition to entailing $\Gamma \Rightarrow \phi$, we want ψ to obey the following three requirements:

1. The solution ψ should be consistent with Γ because an explanation that contradicts known premises is not useful
2. To ensure the simplicity of the explanation, ψ should contain as few variables as possible
3. To capture the generality of the abductive explanation, ψ should be no stronger than any other solution ψ' satisfying the first two requirements

Now, consider a *minimum satisfying assignment* (MSA) of $\Gamma \Rightarrow \phi$. An MSA of a formula φ is a partial satisfying assignment of φ that contains as few variables as possible. The formal definition of MSAs as well as an algorithm for computing them are given in [100]. Clearly, an MSA σ of $\Gamma \Rightarrow \phi$ entails $\Gamma \Rightarrow \phi$ and satisfies condition (2). Unfortunately, an MSA of $\Gamma \Rightarrow \phi$ does not satisfy condition (3), as it is a logically strongest solution containing a given set of variables.

```
abduce(φ, Γ) {
1.   φ = (Γ ⇒ φ)
2.   Set X = find_mus(φ, Γ, free(φ),0)
3.   χ = elim(∀X.φ)
4.   ψ = simplify(χ,Γ)
5.   return ψ
}

find_mus(φ, Γ, V, L) {
6.   If V = ∅ or |V| ≤ L  return ∅
7.   U = free(φ) − V
8.   if( UNSAT (Γ ∧ ∀U.φ))  return ∅

9.   Set best = ∅
10.   choose x ∈ V

11.   if(SAT(∀x.φ)) {
12.    Set Y = find_mus(∀x.φ, Γ, V \ {x}, L−1);
13.    If (|Y|+1 > L) { best = Y ∪ {x}; L = |Y|+1 }
     }
14.   Set Y = find_mus(φ, Γ, V \ {x},L);
15.  If (|Y| > L) { best = Y }

16.  return best;
}
```

Figure 8.1: Algorithm for performing abduction

Given an MSA of $\Gamma \Rightarrow \phi$ containing variables V, we observe that a logically weakest solution containing only V is equivalent to $\forall \overline{V}.\ (\Gamma \Rightarrow \phi)$, where $\overline{V} = \text{free}(\Gamma \Rightarrow \phi) - V$. Hence, given an MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, an abductive solution satisfying all conditions (1)-(3) can be obtained by applying quantifier elimination to $\forall \overline{V}.\ (\Gamma \Rightarrow \phi)$.

Thus, to solve the abduction problem, what we want is a largest set of variables X such that $(\forall X.(\Gamma \Rightarrow \phi)) \wedge \Gamma$ is satisfiable. We call such a set of variables X a *maximum universal subset* (MUS) of $\Gamma \Rightarrow \phi$ with respect to Γ. Given an MUS X of $\Gamma \Rightarrow \phi$ with respect to Γ, the desired solution to the abductive inference problem is obtained by eliminating quantifiers from $\forall X.(\Gamma \Rightarrow \phi)$ and then simplifying the resulting formula with respect to Γ using the algorithm from [101].

Pseudo-code for our algorithm for solving an abductive inference problem defined by premises Γ and conclusion ϕ is shown in Figure 8.1. The `abduce` function given in lines 1-5 first computes an MUS of $\Gamma \Rightarrow \phi$ with respect to Γ using the helper `find_mus` function. Given such a maximum universal subset X, we obtain a quantifier-free abductive solution χ by applying quantifier elimination to the formula $\forall X.(\Gamma \Rightarrow \phi)$. Finally, at line 4, to ensure that the final abductive solution does not contain redundant sub-parts that are implied by the premises, we apply the simplification algorithm from [101] to χ. This yields our final abductive solution ψ which satisfies our criteria of minimality and generality and that is not redundant with respect to the original premises.

LOC analyzed	11,678
Analysis Time	143s
Number of lines changed	76
Annotations placed	7

Figure 8.2: Statistics on the OpenSSH analysis

The function `find_mus` used in `abduce` is shown in lines 6-16 of Figure 8.1. This algorithm is based directly on the `find_mus` algorithm we presented earlier in [100] but excludes universal subsets that contradict Γ. At every recursive invocation, `find_mus` picks a variable x from the set of free variables in φ. It then recursively invokes `find_mus` to compute the sizes of the universal subsets with and without x and returns the larger universal subset. In this algorithm, L is a lower bound on the size of the MUS and is used to terminate search branches that cannot improve upon an existing solution. Therefore, the search for an MUS terminates if we either cannot improve upon an existing solution L, or the universal subset U at line 7 is no longer consistent with Γ. The return value of `find_mus` is therefore a largest set X of variables for which $\Gamma \wedge \forall X.\varphi$ is satisfiable.

8.1.3 Using Abduction to Identify Imprecision Root Causes on Open SSH

In this work, we created a self-contained version of OpenSSH that can be verified automatically, and used abductive inference to identify the missing pieces of information that we needed to add. This can range from simple additional assumptions to rewrites of code pieces to stubbing the behavior of sub-components. Table 8.2 gives a high-level overview of the changes needed for our tool to establish absence of null pointer dereference errors as well as array/buffer overflow or underflow errors.

As mentioned in the last section, the first key challenge is to identify possible small and relevant root causes from a spurious error report. All changes and annotations were identified using logical abduction, and we found this approach to be critical for any non-expert in identifying and remedying relevant precision losses.

To give the reader an understanding of how abduction helps programmers identify relevant annotations for error reports, consider the following sliced and condensed excerpt from OpenSSH. Observe that for keeping this example concise, we manually added one safety property that we want to prove, marked with `static_assert` on line 358 (In our full analysis, all necessary such checks are synthesized automatically).

```
1 /* Fatal messages.  This function never returns. */
2 void
3 fatal(const char *fmt,...)
4 {
5   exit(1);
6 }
7
8 void *
9 xmalloc(size_t size)
```

```
{
  void *ptr;

  if (size == 0)
    fatal("xmalloc: zero size");
  ptr = malloc(size);
  if (ptr == NULL)
    fatal("xmalloc: out of memory (allocating %lu bytes)", (u_long)
       size);
  return ptr;
}

typedef struct {
  unsigned int num_host_key_files;     /* Number of files for host
     keys. */
  int     rhosts_rsa_authentication;    /* If true, permit rhosts RSA
  * authentication. */
  int     rsa_authentication;           /* If true, permit RSA
     authentication. */
  int challenge_response_authentication;
  int password_authentication;
} ServerOptions;




int errno;
/* import */
extern ServerOptions options;
extern char *__progname;
extern uid_t original_real_uid;
extern uid_t original_effective_uid;
extern pid_t proxy_command_pid;



void *
xcalloc(size_t nmemb, size_t size)
{
  void *ptr;

  if (size == 0 || nmemb == 0)
    fatal("xcalloc: zero size");
  if (SIZE_T_MAX / nmemb < size)
    fatal("xcalloc: nmemb * size > SIZE_T_MAX");
  ptr = calloc(nmemb, size);
```

```
54    if (ptr == NULL)
55      fatal("xcalloc: out of memory (allocating %lu bytes)",
56            (u_long)(size * nmemb));
57            return ptr;
58  }
59
60  void
61  xfree(void *ptr)
62  {
63    if (ptr == NULL)
64      fatal("xfree: NULL pointer given as argument");
65    free(ptr);
66  }
67
68
69
70
71  static Authctxt *authctxt;
72  /* message to be displayed after login */
73  Buffer loginmsg;
74
75  /*
76  * Any really sensitive data in the application is contained in this
77  * structure. The idea is that this structure could be locked into
       memory so
78  * that the pages do not get written into swap.  However, there are
       some
79  * problems. The private key contains BIGNUMs, and we do not (in
       principle)
80  * have access to the internals of them, and locking just the
       structure is
81  * not very useful.  Currently, memory locking is not implemented.
82  */
83  struct {
84    Key   *server_key;    /* ephemeral server key */
85    Key   *ssh1_host_key; /* ssh1 host key */
86    Key   **host_keys;    /* all private host keys */
87    int   have_ssh1_key;
88    int   have_ssh2_key;
89    u_char        ssh1_cookie[SSH_SESSION_KEY_LENGTH];
90  } sensitive_data;
91
92  void
93  mm_ssh1_session_id(u_char session_id[16])
94  {
95    Buffer m;
96    int i;
```

```
97
98    debug3("%s␣entering", __func__);
99
100   buffer_init(&m);
101   for (i = 0; i < 16; i++)
102     buffer_put_char(&m, session_id[i]);
103
104   mm_request_send(&m);
105   buffer_free(&m);
106 }
107
108
109 /*
110 * Decrypt session_key_int using our private server key and private
       host key
111 * (key with larger modulus first).
112 */
113 int
114 ssh1_session_key(BIGNUM *session_key_int)
115 {
116   int rsafail = 0;
117
118   if (BN_cmp(sensitive_data.server_key->rsa->n,
119     sensitive_data.ssh1_host_key->rsa->n) > 0) {
120     /* Server key has bigger modulus. */
121     if (BN_num_bits(sensitive_data.server_key->rsa->n) <
122       BN_num_bits(sensitive_data.ssh1_host_key->rsa->n) +
123       SSH_KEY_BITS_RESERVED) {
124       fatal("do_connection:␣%s:␣"
125       "server_key␣%d␣<␣host_key␣%d␣+␣SSH_KEY_BITS_RESERVED␣%d",
126       get_remote_ipaddr(),
127       BN_num_bits(sensitive_data.server_key->rsa->n),
128       BN_num_bits(sensitive_data.ssh1_host_key->rsa->n),
129       SSH_KEY_BITS_RESERVED);
130     }
131     if (rsa_private_decrypt(session_key_int, session_key_int,
132       sensitive_data.server_key->rsa) <= 0)
133       rsafail++;
134     if (rsa_private_decrypt(session_key_int, session_key_int,
135       sensitive_data.ssh1_host_key->rsa) <= 0)
136       rsafail++;
137   } else {
138     /* Host key has bigger modulus (or they are equal). */
139     if (BN_num_bits(sensitive_data.ssh1_host_key->rsa->n) <
140       BN_num_bits(sensitive_data.server_key->rsa->n) +
141       SSH_KEY_BITS_RESERVED) {
142       fatal("do_connection:␣%s:␣"
```

```
143         "host_key␣%d␣<␣server_key␣%d␣+␣SSH_KEY_BITS_RESERVED␣%d",
144         get_remote_ipaddr(),
145         BN_num_bits(sensitive_data.ssh1_host_key->rsa->n),
146         BN_num_bits(sensitive_data.server_key->rsa->n),
147         SSH_KEY_BITS_RESERVED);
148       }
149       if (rsa_private_decrypt(session_key_int, session_key_int,
150         sensitive_data.ssh1_host_key->rsa) < 0)
151         rsafail++;
152       if (rsa_private_decrypt(session_key_int, session_key_int,
153         sensitive_data.server_key->rsa) < 0)
154         rsafail++;
155     }
156     return (rsafail);
157 }
158
159
160 /* session identifier, used by RSA-auth */
161 u_char session_id[16];
162 /* variables used for privilege separation */
163 int use_privsep;
164
165 /* Destroy the host and server keys.  They will no longer be needed.
       */
166 void
167 destroy_sensitive_data(void)
168 {
169     int i;
170
171     if (sensitive_data.server_key) {
172       key_free(sensitive_data.server_key);
173       sensitive_data.server_key = NULL;
174     }
175     for (i = 0; i < options.num_host_key_files; i++) {
176       if (sensitive_data.host_keys[i]) {
177         key_free(sensitive_data.host_keys[i]);
178         sensitive_data.host_keys[i] = NULL;
179       }
180     }
181     sensitive_data.ssh1_host_key = NULL;
182     memset(sensitive_data.ssh1_cookie, 0, SSH_SESSION_KEY_LENGTH);
183 }
184
185 /*
186 * SSH1 key exchange
187 */
188 static void
```

```
189 do_ssh1_kex(void)
190 {
191   int i, len;
192   int rsafail = 0;
193   BIGNUM *session_key_int;
194   u_char session_key[SSH_SESSION_KEY_LENGTH];
195   u_char cookie[8];
196   u_int cipher_type, auth_mask, protocol_flags;
197
198   /*
199   * Generate check bytes that the client must send back in the user
200   * packet in order for it to be accepted; this is used to defy ip
201   * spoofing attacks.  Note that this only works against somebody
202   * doing IP spoofing from a remote machine; any machine on the local
203   * network can still see outgoing packets and catch the random
204   * cookie.  This only affects rhosts authentication, and this is one
205   * of the reasons why it is inherently insecure.
206   */
207   arc4random_buf(cookie, sizeof(cookie));
208
209   /*
210   * Send our public key.  We include in the packet 64 bits of random
211   * data that must be matched in the reply in order to prevent IP
212   * spoofing.
213   */
214   packet_start(SSH_SMSG_PUBLIC_KEY);
215   for (i = 0; i < 8; i++)
216     packet_put_char(cookie[i]);
217
218   /* Store our public server RSA key. */
219   packet_put_int(BN_num_bits(sensitive_data.server_key->rsa->n));
220   packet_put_bignum(sensitive_data.server_key->rsa->e);
221   packet_put_bignum(sensitive_data.server_key->rsa->n);
222
223   /* Store our public host RSA key. */
224   packet_put_int(BN_num_bits(sensitive_data.ssh1_host_key->rsa->n));
225   packet_put_bignum(sensitive_data.ssh1_host_key->rsa->e);
226   packet_put_bignum(sensitive_data.ssh1_host_key->rsa->n);
227
228   /* Put protocol flags. */
229   packet_put_int(SSH_PROTOFLAG_HOST_IN_FWD_OPEN);
230
231   /* Declare which ciphers we support. */
232   packet_put_int(cipher_mask_ssh1(0));
233
234   /* Declare supported authentication types. */
235   auth_mask = 0;
```

```
236   if (options.rhosts_rsa_authentication)
237     auth_mask |= 1 << SSH_AUTH_RHOSTS_RSA;
238   if (options.rsa_authentication)
239     auth_mask |= 1 << SSH_AUTH_RSA;
240   if (options.challenge_response_authentication == 1)
241     auth_mask |= 1 << SSH_AUTH_TIS;
242   if (options.password_authentication)
243     auth_mask |= 1 << SSH_AUTH_PASSWORD;
244   packet_put_int(auth_mask);
245
246   /* Send the packet and wait for it to be sent. */
247   packet_send();
248   packet_write_wait();
249
250   debug("Sent␣%d␣bit␣server␣key␣and␣%d␣bit␣host␣key.",
251   BN_num_bits(sensitive_data.server_key->rsa->n),
252   BN_num_bits(sensitive_data.ssh1_host_key->rsa->n));
253
254   /* Read clients reply (cipher type and session key). */
255   packet_read_expect(SSH_CMSG_SESSION_KEY);
256
257   /* Get cipher type and check whether we accept this. */
258   cipher_type = packet_get_char();
259
260   if (!(cipher_mask_ssh1(0) & (1 << cipher_type)))
261     packet_disconnect("Warning:␣client␣selects␣unsupported␣cipher.");
262
263   /* Get check bytes from the packet.  These must match those we
264   sent earlier with the public key packet. */
265   for (i = 0; i < 8; i++)
266     if (cookie[i] != packet_get_char())
267       packet_disconnect("IP␣Spoofing␣check␣bytes␣do␣not␣match.");
268
269     debug("Encryption␣type:␣%.200s", cipher_name(cipher_type));
270
271   /* Get the encrypted integer. */
272   if ((session_key_int = BN_new()) == NULL)
273     fatal("do_ssh1_kex:␣BN_new␣failed");
274   packet_get_bignum(session_key_int);
275
276   protocol_flags = packet_get_int();
277   packet_set_protocol_flags(protocol_flags);
278   packet_check_eom();
279
280   /* Decrypt session_key_int using host/server keys */
281   rsafail = PRIVSEP(ssh1_session_key(session_key_int));
282
```

```
283   /*
284   * Extract session key from the decrypted integer.  The key is in
          the
285   * least significant 256 bits of the integer; the first byte of the
286   * key is in the highest bits.
287   */
288   if (!rsafail) {
289     (void) BN_mask_bits(session_key_int, sizeof(session_key) * 8);
290     len = BN_num_bytes(session_key_int);
291     if (len < 0 || (u_int)len > sizeof(session_key)) {
292       error("do_ssh1_kex:␣bad␣session␣key␣len␣from␣%s:␣"
293       "session_key_int␣%d␣>␣sizeof(session_key)␣%lu",
294       get_remote_ipaddr(), len, (u_long)sizeof(session_key));
295       rsafail++;
296     } else {
297       memset(session_key, 0, sizeof(session_key));
298       BN_bn2bin(session_key_int,
299                 session_key + sizeof(session_key) - len);
300
301                 derive_ssh1_session_id(
302                 sensitive_data.ssh1_host_key->rsa->n,
303                                        sensitive_data.server_key->rsa
                                           ->n,
304                                        cookie, session_id);
305                                        /*
306                                        * Xor the first 16 bytes of the
                                             session key with the
307                                        * session id.
308                                        */
309                                        for (i = 0; i < 16; i++)
310                                          session_key[i] ^= session_id[
                                             i];
311     }
312   }
313   if (rsafail) {
314     int bytes = BN_num_bytes(session_key_int);
315     u_char *buf = xmalloc(bytes);
316     MD5_CTX md;
317
318     logit("do_connection:␣generating␣a␣fake␣encryption␣key");
319     BN_bn2bin(session_key_int, buf);
320     MD5_Init(&md);
321     MD5_Update(&md, buf, bytes);
322     MD5_Update(&md, sensitive_data.ssh1_cookie,
          SSH_SESSION_KEY_LENGTH);
323     MD5_Final(session_key, &md);
324     MD5_Init(&md);
```

```
325       MD5_Update(&md, session_key, 16);
326       MD5_Update(&md, buf, bytes);
327       MD5_Update(&md, sensitive_data.ssh1_cookie,
              SSH_SESSION_KEY_LENGTH);
328       MD5_Final(session_key + 16, &md);
329       memset(buf, 0, bytes);
330       xfree(buf);
331       for (i = 0; i < 16; i++)
332         session_id[i] = session_key[i] ^ session_key[i + 16];
333     }
334     /* Destroy the private and public keys. No longer. */
335     destroy_sensitive_data();
336
337     if (use_privsep)
338       mm_ssh1_session_id(session_id);
339
340     /* Destroy the decrypted integer.  It is no longer needed. */
341     BN_clear_free(session_key_int);
342
343     /* Set the session key.  From this on all communications will be
           encrypted. */
344     packet_set_encryption_key(session_key, SSH_SESSION_KEY_LENGTH,
           cipher_type);
345
346     /* Destroy our copy of the session key.  It is no longer needed. */
347     memset(session_key, 0, sizeof(session_key));
348
349     debug("Received␣session␣key;␣encryption␣turned␣on.");
350
351     /* Send an acknowledgment packet.  Note that this packet is sent
           encrypted. */
352     packet_start(SSH_SMSG_SUCCESS);
353     packet_send();
354     packet_write_wait();
355
356
357     for (i = 0; i < options.num_host_key_files; i++) {
358       static_assert(sensitive_data.host_keys[i] == NULL);
359     }
```

Observe that even in this sliced and self-contained code fragment, it is far from clear why the analysis is reporting this error and what information is missing or where precision was lost. Using logical abduction, our tool automatically identifies the following fact:

‘‘ If function destroy_sensitive_data() sets every element of sensitive_data.host_keys in range[0, options.num_host_key_files] to NULL, this will prove the assertion.’’

More formally, we identify a simplest and most general solution to the abduction problem as:

$$\psi := \forall i.(0 \leq i < options.num\_host\_key\_files \wedge \text{call}(destroy\_sensitive\_data)) \rightarrow (\text{array}(\text{sensitive\_data.host\_keys}, i) = 0)$$

While the desired property clearly holds for the loop in `destroy_sensitive_data()`, the verification tool, for internal reasons that are obscure to the tool user, fails to understand this loop properly. In this case, this is clearly the case, and after annotating this missing piece of information, the assertion is now provable. Specifically, we add the following simple annotation to line 176:

```
assume( (!(0<=_t && _t <= i)) || ( sensitive_data.host_keys[_t] == NULL));
```

where the syntax `_t` marks `_t` as an universally quantified variable.

Observe that logical abduction helped us immediately pin down the root cause of this spurious report without any need for the user to be familiar with the internal reasoning and limitations of the analysis tool used.

8.1.4 Code Changes and Annotations

In order to complete the verification of memory safety properties, we also added a few key annotations, as well as modified some lines of code in order to facility automated analysis of the code. This mostly involved expressing global object invariants as annotations, as well as replacing built-in macros that copy memory contents with explicit store statements.

Here is a code snippet from `clientloop.c` to illustrate this:

```
1
2  // Annotated global invariants
3  assume(options.num_send_env <= MAX_SEND_ENV);
4  assume(options.num_send_env >=0);
5  assume(env_len*sizeof(char*) <= buffer_size(env));
6
7  /* Transfer any environment variables from client to server */
8  if (options.num_send_env != 0 && env != NULL) {
9          int i, j, matched;
10         char *name, *val;
11
12         debug("Sending␣environment.");
13         for (i = 0; i<env_len; i++) {
14                 // here we replaced pointer arithmetic used in
15                 // the original code with an explicit array reference
                       style
16
17                 /* Split */
18                 name = env[i];
19
```

```
20                      if ((val = foo(name, '=')) == NULL) {
21                              xfree(name);
22                              continue;
23                      }
24
25                      *val++ = '\0';
26
27                      matched = 0;
28                      for (j = 0; j < options.num_send_env; j++) {
29                              if (match_pattern(name, options.send_env[j]))
                                    {
30                                      matched = 1;
31                                      break;
32                              }
33                      }
34                      if (!matched) {
35                              debug3("Ignored␣env␣%s", name);
36                              xfree(name);
37                              continue;
38                      }
39
40                      debug("Sending␣env␣%s␣=␣%s", name, val);
41                      channel_request_start(id, "env", 0);
42                      packet_put_cstring(name);
43                      packet_put_cstring(val);
44                      packet_send();
45                      xfree(name);
46              }
47      }
```

Note that the key object invariants relating global values are annotated in order to allow the successful analysis of this code segment.

8.2 Dark Corners

Sound static program analysis promises exhaustive detection of many classes of program errors and, ideally, verification that the program is free of these errors. Despite decades of effort investing in developing powerful static analyses, we are still far from having a static analysis that can analyze existing programs precisely enough to make verifying the absence of important errors (such as memory safety vulnerabilities) feasible in practice. The critical issue is the lack of precision that such analyses inevitably encounter when analyzing the programs that occur in practice. This lack of precision causes either 1) unacceptable numbers of false positive alarms or 2) the use of unsound techniques that may leave errors uncovered.

Our hypothesis is that, in existing programs, only a small percentage of the code (the code's dark corners) is responsible for this lack of precision. If this hypothesis is true, it opens up the pos-

sibility that we are much closer to a practical program analysis for verifying important security (and potentially other properties) than it currently appears. Our work on analyzing the Java system libraries in DroidSafe [1] supports this. In many cases, small changes to the library code significantly increased the precision of the analysis.

We believe that once the small percentage is identified that there are reasonable techniques for addressing the complex code. For example: (1) Applying more expensive analysis techniques to this code; (2) Making manual changes to the code and/or adding annotations/dynamic checks to reduce the complexity; (3) Replacing the code with similar code from other projects (e.g., DARPA MUSE); (4) Providing an alternative implementation that is easier to analyze but can be shown to match the existing implementation (with respect to memory safety).

We investigated this hypothesis on a set of widely used open source C programs from coreutils. Specifically mv.c and chroot.c Our goal was to create versions of the programs for which we can verify memory safety (out of bounds accesses), null pointer dereferences, and the use of uninitialized variables.

We used a focused version of the Compass tool [102, 103, 92] (CONCORD) adapted and streamlined for robustness and coverage for this study.

8.2.1 Concord Features

Concord analyzes C programs and processes some additional functions to aid in the analysis. It can check assertions added by the programmer and can automatically check for buffer overflows, null pointer dereferences, and the use of uninitialized variables.

Concord supports a number of special functions. These are most commonly used in library routine summaries though there are use cases (primarily for `static_assert`) in the application as well.

The primary functions supported by Concord are:

- `void static_assert (<expr>)` - Statically checks the specified expression.
- `void assume (<expr>)` - Assume that the specified expression is true
- `<type> set\_nonnull_<type>` - Function that returns a value that is non-null (non-zero) for the specified type. This function must be declared with the correct type. Concord does not operate correctly if there are type mismatches between the return value of `set_nonnull` and the variable it is assigned to.
- `int unknown()` - Returns an arbitrary initialized integer value
- `void buffer\_safe (void *buffer, int offset)` - Statically checks whether or not it is safe to index into `buffer` at `offset`
- `void assume\_size (void *buffer, int len)` - Assume that the size of `buffer` is `len`

8.2.2 Coding practices and analysis

We discovered a number of coding practices that make a significant difference to the analysis. In most cases, the alternative easier to analyze version is basically equivalent in terms of coding effort and efficiency.

8.2.2.1 Obscure loop iterations / indexing

Concord attempts to find invariants as part of analyzing loops. One of its main approaches is to attempt to find a relationship between the a loop counter (K) and various loop variables. This works well when the loop is straightforward (such as a loop over an array of values), but may fail to provide interesting information or timeout when the relationship is obscure.

For example the C library function `getopt()` processes command line arguments. Each call to `getopt()` returns the next valid option character. A typical loop that invokes `getopt()` looks like:

```
1   while ((c = getopt (argc, argv,"bfint:uvS:T") != -1) {
2     // switch on option character
3     switch (c)
4     ...
5   }
```

To support this usage model, the internals of `getopt()` uses the external variable `optind` to keep track of the current argument and the static variable `nextchar` to keep track of the next argument and next option character within that argument. These variables are updated on each call. A greatly simplified version of `getopt()` shows roughly how this works:

```
1  int getopt (int argc, char **argv,  char *options) {
2  {
3    if (nextchar == NULL) || (*nextchar == '\0') {
4
5      if(optind == 0) optind = 1;
6          while ((optind < argc) && (argv[optind][0] != '-'))
7          optind++;
8      if (optind >= argc) return -1;
9      nextchar = argv[optind] + 1;
10   }
11   return (*nextchar++);
12 }
```

There is no straightforward relationship between the the loop counter and any of the variables. In order to create a simple interface for the caller, `getopt()` turns what might naturally be two nested loops into a single loop which significantly complicates the analysis. In this case, Concord times out while attempting to simplify the loop.

If this were not a utility routine it could be implemented in a much more straightforward fashion as two nested loops. The first loop is over each of the arguments and then within each option

argument (those that begin with a dash), a loop over each option character in the argument. For example:

```
1    for (i = 0; i < argc; i++) {
2      char *arg = argv[i];
3      if (*arg++ == '-') {
4        while (*arg != 0) {
5          ch = *arg;
6          switch (c)
7          ...
```

This version works fine in Concord. (and is arguably just as easy to use as the original) But some of the complexity is no longer hidden in a library routine. This could be resolved by creating two library routines (one to loop through the arguments and the other to process an argument).

Another approach is to provide a simpler summary of the library method rather than analyzing the method itself. This has the downside of not proving correctness of the library routine, but can create a much more analyzable version for the application. In this case, we replaced `getopt()` by:

```
1  int getopt (int argc, char **argv,  char *options) {
2
3    static_assert (argv != NULL);
4
5    // optind is guaranteed to point into argv */
6    optind = unknown();
7    assume ((optind >= 1> && (optind < argc));
8
9    // optarg is a pointer to the value for an argument.  An argument
       value
10   // may either immediately follow the option character or be in the
11   // next argument
12   optarg = unknown();
13
14   // Each of the strings in argv should be valid
15   int ii;
16   for (ii = 0; ii < argc; ii++) {
17     buffer_safe (argv, ii);  // check that ii is a safe index into
         argv
18     check_str_nn (argv[ii]);
19   }
20
21   // The return value is either a character or -1
22   int rval = unknown();
23   assume ((rval == -1) || ((rval > 0)  && rval < 255));
24   return rval;
25 }
```

This code checks all of its arguments for validity and sets up reasonable return values. This allows the caller to be verified by Concord.

This version is also handled correctly by Concord and shows that there are no errors in the client (mv, chroot, etc)

Unfortunately, the summary approach may miss some nuanced problems with the use of `getopt()` such as when there are arguments associated with options. The `optarg` value is only set when such an argument is encountered. The summary has to set `optarg` to a valid value on each call (because it does not know which calls will have an option value) It is possible that a caller may access `optarg` when it is not set which would lead to a memory error.

This problem can be handled when options are handled as two nested loops.

```
1   for (i = 0; i < argc; i++) {
2     char *arg = argv[i];
3     if (*arg == '-') {
4       arg++;
5       while (*arg++) {
6         ch = *arg;
7         switch (c){
8         case 'b':
9           make_backups = true;
10          break;
11        case 'f':
12          x.interactive = I_ALWAYS_YES;
13          break;
14        case 'i':
15          x.interactive = I_ASK_USER;
16          break;
17        case 'n':
18          x.interactive = I_ALWAYS_NO;
19          break;
20        case 't':
21          if (*arg) {
22            target_directory = arg;
23          } else { // directory is in the next argument
24            target_directory = argv[i++]
25          }
26          // move arg pointer to the end of the argument to break the
                loop
27          arg += strlen(arg);
28          break;
29        case 'T':
30          no_target_directory = true;
31          break;
32        case 'u':
33          x.update = true;
34          break;
35        case 'v':
36          x.verbose = true;
37          break;
```

```
38          case 'S':
39            make_backups = true;
40           // backup_suffix_string = optarg;
41            break;
42          case_GETOPT_HELP_CHAR;
43          case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
44          default:
45            usage (EXIT_FAILURE);
46          }
47          ...
```

8.2.2.2 Additional Checks

Any static analysis can be confused by complex relationships between variables. It may be that at a particular point in the program, that a certain condition must be true and the program is safe to assume that. The static analysis may not be able to reason about this and may produce a false positive.

One simple solution to this problem is to add a redundant check for the condition where it is assumed. For example, `getopts()` handles command line options that take an argument. If the argument is missing, `getopts()` will, be default, issue an error message and terminate. The application can thus safely assume that the argument is not null. This correspondence would be extremely difficult to determine analyzing the `getopt()` code and impossible when using the summary.

However, it is easy to avoid the problem by simply adding an additional null check for the argument at its point of use. This was necessary only a few times in our example programs.

8.2.2.3 Pointer Arithmetic

Complex pointer arithmetic can be difficult to analyze and lead to innacuracies in analysis. In many cases, this can be replaced by an array reference. This can make the relationship between the loop variable and the array reference more explicit.

This is the change that we made in `clientloop.c` to allow the analysis to remove false positives.

8.2.3 Verification Approach

We verified several coreutils programs for memory safety as part of the project. The verification concentrated on the main file of the program. Calls to methods in the standard C library and the coreutils libraries were not included. These methods were summarized with respect to memory safety. Each summary checks the validity of each of the arguments to the function and sets any return values to the correct range of valid values. For example, the summary of `stat()` is:

```
1 int stat (char const *file, struct stat *st) {
2
```

```
3     // Make sure arguments are not null
4     static_assert (file != NULL);
5     static_assert (st != NULL);
6
7     // Initialize fields to non-zero ints
8     st->st_dev = set_nonnull_int();
9     st->st_ino = set_nonnull_int();
10    st->st_mode = set_nonnull_int();
11
12    // Return normally or error
13    if (set_nonnull_int() == 1)
14      return 0;
15    else { // simulated error
16      errno = set_nonnull_int();
17      return -1;
18    }
19  }
```

Note that this initializes fields to non-zero values. This ensures that accesses to these field will not result in uninitialized read errors.

8.2.4 Concord limitations

While we addressed a number of limitations in Concord, there is additional work that would allow a wider range of programs to be verified and ease the work of doing so. Some of these are listed below.

Valid Strings. Null terminated strings are a very basic type in C. It is difficult to define these efficiently using the building blocks currently available in Concord. A valid null-terminated string should consist of an array of initialized characters followed by the null character. The array of characters (including the null) should be less than or equal to the size of the buffer that contains them. In concord, such a check could be coded something like:

```
1  void valid_string (char *str) {
2    int i;
3    for (int i = 0; str[i] != '\0'; i++)
4      char ch = str[i];
5    static_assert (buffer_size (str) > i);
6  }
```

Unfortunately, it is not as easy to specify that an unknown string (such as one read from a command line argument or a file) is a valid one. And Concord doesn't currently have a mechanism to assign an arbitrary size to a buffer (such as might be returned from `getenv()`)

Given the prevalence of strings and string manipulations in C, it seems worthwhile to support them directly in Concord. This could be accomplished by adding some new functions. The function

`check_str()` would check for a valid string, the function `set_str()` would mark the string as valid or NULL, and the function `set_str_nonnull()` would mark the strings as valid and not null. This would allow functions that accept strings as arguments to perform checks on their inputs and return valid outputs. It would also make it straightforward to provide summaries for the standard C string functions.

Object invariants. Object invariants are are constraints on an object that should be true at entry/exit to all of the public methods of the object. The valid string checks described above are a special case of these. Object invariants can simplify static checking by providing assumptions that will always hold over an object of a particular type. If Concord were to be enhanced with support for object invariants (possibly over C structures), it would be easier to verify more complex programs.

Assumes. Currently Concord does not propagate information about variables specified by assumes (or implied by conditional statements) as cleanly as it propagates sets of possible values. This can lead to false positives when the necessary information to verify a property is available but not fully propagated.

Loop Invariants. Concord attempts to learn relationships between the iterations of a loop and any variables that are manipulated in the loop. Allowing loop invariants to be specified would make it possible to handle more complex loops.

Varargs. Concord doesn't currently support variable argument lists, but this would be any easy enhancement.

Memory Management Checks. With respect to memory safety, Concord checks for out-of-bounds buffer accesses, reading uninitialized values, and null dereferences. It could be enhanced to check for common memory management errors such as Memory leaks, double frees, etc.

Non-deterministic. Due to low level implementation choices, some of the basic data structures used within Concord do not yield deterministic results (one run may timeout or show false positives while a subsequent run does not). It is a straightforward fix to change the underlying sets, lists, and maps to have a repeatable order.

8.2.5 Debugging

Debugging false positives (and differentiating between false positives and real problems) when verifying code can be challenging to the non-expert. However, as part of this project we discovered a relatively straightforward process for quickly finding the source of problems.

In some sense, debugging a verification error is very much like debugging a normal coding error. The static analysis indicates that a particular operation is not necessarily correct (e.g., it is dereferencing a possibly null variable or reading an uninitialized variable). If the assertion were checked at run-time, it would fail in a very similar fashion. A standard debugging technique would be to add debug statements to earlier points in the program to determine where the unexpected value came from.

A very similar approach can be taken with Concord. One can add assertions earlier in the data-flow of the variable making the same check. Iteratively re-running the analysis and adding additional assertions can quickly narrow down the problem (perhaps using a rough binary search). Once the root cause of the problem is found, it is usually pretty straightforward to fix.

This basic approach can be applied more generically as well. It is useful to add assertions at standard program points (such as function entry and exit points). They can help make the assumptions of the function and its results more clear. The systematic presence of such asserts will often uncover problems much closer to the source. We found that to be the case when working with our library summaries. Calls to the libraries often triggered failures quite close to the actual problem that would otherwise have appeared much later.

Not surprisingly, this approach works best when dealing with false positives. It is much less helpful in debugging problems that cause timeouts in the solver.