Refactoring Programs to Secure Information Flows

Scott F. Smith and Mark Thober

The Johns Hopkins University

{scott,mthober}@cs.jhu.edu

Abstract

Adding a sound information flow security policy to an existing program is a difficult task that requires major analysis of and changes to the program. In this paper we show how refactoring programs into distinct components of high and low security is a useful methodology to aid in the production of programs with sound information flow policies. Our methodology proceeds as follows. Given a program with no information flow controls, a program slicer is used to identify code that depends on high security inputs. High security code so identified is then refactored into a separate component, which may be accessed by the low security component via public method calls. A security policy that labels input data and checks the output points can then enforce the desired end-to-end security property. Controlled information releases can occur at explicit declassification points if deemed safe. The result is a well-engineered program with explicit interfaces between components of different security levels.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks, Input/output ; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and interfaces; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection.

General Terms Design, Languages, Security.

Keywords Information flow, refactoring, declassification, slicing.

1. Introduction

Language-based information flow security [4, 7, 20] focuses on ensuring the confidentiality of data in programs. High security inputs should not affect low security outputs, whether directly (such as passing a secure value to a low output) or indirectly (such as a conditional on a secure value that affects a low output). This is known as a *noninterference* property [9]. In practice, such a property is too restrictive, since real programs will leak some amount of sensitive information (e.g. a boolean comparison with a password could be output to the screen even though the password is a secret). Such information release is allowed through an explicit *declassification* [16]. In this setting, application developers must declare the security levels of the input and output points, and specify any declassifications in the source code; some information flow systems require further code annotations to provide security.

Information flow security models the flow of information through a program. This differs from an access control model which governs the authorization for actions. Other forms of security, such as process privileges and stack inspection, are out of the scope of information flow security. We make the assumption that the underlying platform is trusted and that secure IO channels are indeed secure.

Most information flow research has focused on proving formal properties about small languages [28, 12, 3]. There have been a few efforts focusing on practical incorporation of information flow into programming systems [15, 17], but these systems are still several steps away from practical feasibility: they place a significant overhead on the programmer due to the need to place a large number of annotations in the program.

Several authors have observed how programs naturally segment for information flow security purposes. Li and Zdancewic [13] observe that *"noninterfering programs can be factored into a 'high security' part and a 'low security part.' "* Functions that depend on high-level data require *"downgrading policies"* in order to allow a declassified flow from high to low. Amtoft and Banerjee [2] show how forward slicing can be used to remove code that depends on a high-level variable `h` from a program P, and produce a program P_0. *"Then a user, wanting to compute the low variables but (for security reasons) not given clearance to view* `h`*, could be given* P_0 *to run, rather than* P*."*

Software refactoring and restructuring has recently been a topic of much research [14]. However, this research has focused on software design issues, such as compatibility, reusability, and efficiency, and not on security. The point of this paper is how the technique of refactoring can be applied for security purposes: refactoring programs into high- and low-security components [24] will improve code security in practice. Refactoring out the high-security data puts all the code operating on security-critical data, which is usually a very small portion of an application, in its own component, where it can be subject to significantly higher standards of correctness in coding, testing, and verification. This paper focuses on the challenges and benefits of performing such refactorings on real programs. We illustrate our ideas via examples, including a refactoring of a portion of the OpenSSH client code [30]. One important aspect of refactoring is only sound refactorings must be performed, *i.e.* refactorings which do not change the meaning of the programs.

Figure 1 diagrams how refactoring transforms a program by segmenting it into high- and low- security components, with an access channel between the two portions. The low component will only take low inputs and produce low outputs, although it may accept declassified flows from the high component. The high component accepts both high and low inputs, and may produce only high outputs. For strict *noninterference* [9], there will be no flows from the high portion to the low portion.

By isolating high security code portions, programmers can more easily add information flow controls to a program, as the component boundaries are now the clear places where high security data affects low security data. Just as important is the fact that application developers can now be confident in a program's security policies by observing explicit interactions between security levels. Since the high security code is placed in its own component, developers need only study the high security code and its component interfaces to ensure security; this means a huge reduction in the amount of code that must be inspected. Realistic programs generally only have a small portion of the code performing high security operations; for example, some Microsoft groups have noted that less than 20 percent of their code is security-critical [8], and some experiments with privilege separation have shown there is approximately one privileged call in 2,800 lines of source code [5]. By refactoring out this critical code, programmers will be able to put more focus on it. And, a refactoring that produces a large high security component may indicate erroneous design [23].

The rest of this paper is structured as follows. Our basic refactoring methodology appears in section 2. Section 3 presents an example that refactors part of the SSH client for security of private keys. This leads to some more complex issues for refactoring real programs, which is discussed in section 4. We discuss related work in section 5.

2. Refactoring Methodology

We now address the core concepts and challenges involved in refactoring for information flow security. The refactoring entails three main tasks: first, identifying sensitive code; second, performing a series of sound refactorings which produces a new component containing all of the high-security code; and third, potentially adding explicit information flow controls, likely in the form of explicit declassifications. Developers can take a program that does not have any information flow security and iterate through the refactoring tasks to produce a secure program.

In this paper we take an IO-centric approach to information flow: we assume that all IO channels are labeled as either high- or low- security input or output, and that these are the only data label points in the program. The advantages of this approach are discussed in [21]: effective tracking of information in programs begins and ends at the program's boundaries, and so only those points need to be explicitly labeled; all other labels can be inferred. Information flow security must emphasize clear boundaries and rigorously control information across the boundaries.

We use an informal framework for our examples as follows. The examples use *C* and *C++* code due to our focus on OpenSSH; the methodology should apply to other languages however. We assume a labeling for input channels, such that low inputs label data *low*, and high inputs label data *high*. In a program with sound information flow, data labeled *high* is not output on low output channels, or out of the high-security component. We assume an addition to the language of the statement `Declassify(e,`*high*`)`, that removes the high label from the expression `e`. We assume these labels are properly tracked using a sound static information flow analysis that we do not specify here. For simplicity, this paper considers only high and low security levels. Although some additional complications arise, we believe a similar methodology will apply to programs with multiple security levels.

We illustrate the basic refactoring tasks with an oversimplified example of a login routine. The initial program is in Figure 2. This program inputs a username and password from the keyboard, then attempts to authenticate the user by comparing the username and password with those on the file system.

In this example, the high security data is the username and password stored on the system, which should not leak to public output channels. In the following sections, we show how to identify the high security portion of this code for refactoring, perform the

```
int main() {
 char uname[15];
 char passwd[15];
 char* syspasswd;

 scanf("%s",uname);
 scanf("%s",passwd);
 syspasswd = getPasswd(uname,"/etc/passwd");
 if (!strcmp(syspasswd,passwd)) {
  printf("access granted");
 } else {
  printf("access denied");
  exit(0);
 }
 /* Execution proceeds */
}

char* getPasswd(char* uname, char* file) {
 /* Returns the password for uname in file */
}
```

Figure 2. Login Code

refactoring, and add information flow controls to the refactored program.

2.1 Identifying Sensitive Code

To refactor out high security portions of a program, one must identify where high security data flows through a program. Thus, identification begins at high security input points and continues to wherever this data flows. We adopt a program slicing terminology to the identification task, as slices are natural lines along which to refactor programs [27].

A static program slicer is specified as follows. Given a language grammar and an operational semantics, let P be a valid program in the language. Slicing criteria are defined for forward and backward slices.

DEFINITION 2.1. *A* ***forward slicing criterion*** C_{in} *of a program P is a pair* (l, v_{in})*, where l is a line number in P and* v_{in} *is an input at l in P. A* ***backward slicing criterion*** C_{out} *of a program P is a pair* (l, v_{out})*, where l is a line number in P and* v_{out} *is an output at l in P.*

The relations *affects* and *affected* define the statements that either affect or are affected by slicing criteria.

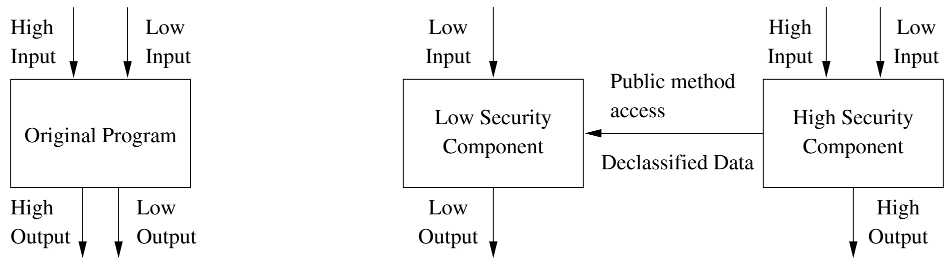
DEFINITION 2.2. *Given a program P, and criterion* $C_{in} = (l, v_{in})$*. For any statement* `s`*,* $\mathit{affected}(P, C_{in}, \mathtt{s})$ *is true if any change in the value of* v_{in} *may cause a change in the value at* `s`*, and false otherwise.*

DEFINITION 2.3. *Given a program P, and criterion* $C_{out} = (l, v_{out})$*. For any statement* `s`*,* $\mathit{affects}(P, C_{out}, \mathtt{s})$ *is true if any change at* `s` *may cause a change in* v_{out}*, and false otherwise.*

A forward slice of a program P with respect to criterion C_{in} is a subset of statements and expressions of P that is directly or indirectly affected by the value at the slicing criterion.

DEFINITION 2.4. *For program P and criterion* C_{in}*, a forward slice* $S_f(P, C_{in}) = \{\mathtt{s} | \mathit{affected}(P, C_{in}, \mathtt{s})\}$.

A backward slice of a program P with respect to criterion C_{out} is a subset of statements and expressions of P that directly or indirectly affect the value at the slicing criterion.



(a) Original Program (b) Refactored Program

Figure 1. Program Refactoring to High and Low Security Portions

DEFINITION 2.5. *For program P and criterion C_{out}, a backward slice $S_b(P, C_{out}) = \{\mathtt{s} | \mathit{affects}(P, C_{out}, \mathtt{s})\}$.*

Note that forward and backward slices are not necessarily executable programs. A *program slicer* is any analysis that soundly computes forward and backward program slices when given a slicing criterion.

A slicer satisfying the above definitions is needed to determine all affected code, so possible leaks of information flows will not be missed. Either a program slicer using PDGs [10, 19] or an information flow type inference system [17, 21] could be used to infer high and low slices. PDG-based slicers have the advantage of being flow-sensitive, taking the order of data operations into account, but have limited expressiveness on higher-order programs. This paper is neutral to whether a slicer or a type inference system has calculated the slices; the examples in this paper have been sliced by hand.

Using a forward slice from a high security input creates a *high slice*, i.e. all of the code that this high input reaches. A backward slice from a low security output creates a *low slice*, i.e. all of the code that affects the low output. This paper focuses on high security inputs, so the concern is with forward slicing from a high input. Figure 3 shows all of the code affected by the high security inputs from the system password file. The slicing criterion is the input from the `getPasswd` method (line 19) called on line 8. The slice is indicated by highlighted statements.

In some programs with both high and low security data, the initially calculated high slice may be too large, possibly including the entire program. This is of course undesirable, and it indicates either that there is an error or that declassifications of high-security data at some points are justified.

In the login example, the screen (`System.out`) is a low security output, so high security data here affects a low output. This leak is however required for proper function of password-based authentication, and so is a point where declassification is justified. So, once the comparison is done, the high security data can be declassified, so it does not taint the rest of the program. Therefore, the code after the conditional at line 9 can remain part of the low security code. In the presence of existing declassifications, computation of the high slice stops wherever a declassification of the tracked flow occurs.

2.2 Refactoring

Once a high slice has been identified, the next step is to refactor it into a new component. Note that in some cases it is necessary for the high component to contain code that is not in the high slice, such as variable declarations and some low input channels. This

```
1:  int main() {
2:    char uname[15];
3:    char passwd[15];
4:    char* syspasswd;
5:
6:    scanf("%s",uname);
7:    scanf("%s",passwd);
8:    syspasswd = getPasswd(uname,"/etc/passwd");
9:    if (!strcmp(syspasswd,passwd)) {
10:     printf("access granted");
11:   } else {
12:     printf("access denied");
13:     exit(0);
14:   }
15:   /* Execution proceeds */
16: }
17:
18: char* getPasswd(char* uname, char* file) {
19:   /* Returns the password for uname in file */
20: }
```

Figure 3. Login Code with High Slice

does not introduce any security risks, as we only need to guarantee that all of the high slice is in the high component.

Figure 4 shows the refactored login code. Here we are refactoring to *C++* to take advantage of object-oriented language features. The class `Login` is created with a field for holding the name of the file. The authentication of the user to the system has been moved to this new class and is accessible through the public method `authenticate`. The `getPasswd` method becomes a private method to the class, as it is needed only in the high security context. The `syspasswd` string is eliminated, as it is no longer needed.

High security code in the refactored program is highlighted. In the low component, high security operations have been reduced to calls to public methods in the high component.

Refactoring is best accomplished in a *maximal munch*-style, taking the largest possible code block that contains only high code and refactoring it into a method. This may be infeasible when high code is tightly interwoven with low code, requiring more complex refactorings. Yet we contend that most programs are written in a manner that performs high security operations in chunks, e.g. reading from a sensitive file or setting up a high security data structure. Our SSH example confirms this (section 3). Code blocks refactored from code portions that remain low are put into `public`

High Component:

```
class Login {
public:
  Login(char* f) { file = f; }

  bool authenticate(char* uname, char* passwd) {
    bool success = false;
    if (!strcmp(getPasswd(uname),passwd)) {
      success = true;
    }
    return Declassify(success,high);
  }
private:
  char* file;
  char* getPasswd(char* uname) {
    /* Returns the password for uname in file */
  }
};
```

Low Component:

```
int main() {
  char uname[15];
  char passwd[15];
  Login l("etc/passwd");

  scanf("%s",uname);
  scanf("%s",passwd);
  if (l.authenticate(uname,passwd)) {
    printf("access granted");
  } else {
    printf("access denied");
    exit(0);
  }
  /* Execution proceeds */
}
```

Figure 4. Refactored Login Code

methods in the new class, and calls to these methods are put in the low code where the refactored code was removed.

In addition to refactoring chunks of code, entire functions may be moved to the high-security component. Functions that are called solely by high code portions may be moved to high-security classes as `private` methods. In a similar manner, variables and data structures holding high security data may be placed as private fields in the new class.

The refactoring process can be improved with automated assistance. Semi-automated refactorings are transformations that ensure the correctness and behavior-preservation of the change [14, 26]. For example, an automated **MoveMethod** refactoring moves a method from one class to another, checking for naming clashes and other standard problems. Fully automated refactorings are not feasible, however, because they often result in code that is more difficult to understand [14, 6]. Furthermore, the choice of which refactorings to perform must be decided by the developer. For example, the high slice through a method body may not include initialization code, yet the entire method may be moved to the high component.

2.3 Adding Information Flow Controls

Adding information flow controls to programs that previously had none can be challenging, and may require significant additional effort by the developer. Data must properly be labeled at input points, and checks need to be placed at output points. This is a crucial step in the process, as systems can only guarantee information flow security that satisfies the labeling and checking mechanisms; they cannot guarantee that the labels and checks are themselves the correct policy. It is up to the programmer to write the correct policy concerning IO points.

Improper placement of declassification statements can cause security leaks in a program if sensitive data is inadvertently declassified. Deciding that declassification is "safe" is a difficult task that requires knowledge of the program implementation. Section 4.2 describes the challenges of aiding programmers in placing declassifications, and discusses some principles for overcoming these problems using refactoring.

For our login example in Figure 4, the `getPasswd` method must label the data coming in on the file input stream as *high*. The result of the boolean comparison of the user's password on the system with the challenge password is declassified in `authenticate`; this result can safely be passed to the public code portion, which passes the result to a low output channel.

3. Refactoring SSH

In order to see some of the subtle issues arising from refactoring out the high slice, we study the OpenSSH client implementation. We focus on a specific portion of authentication in the client, namely the combined rhosts-RSA authentication. We first provide a short discussion of how this authentication method works, and how information flow security is useful. Note this authentication method is specific only to SSH-1. A segmenting of the OpenSSH server into different components for better security was carried out in [18]; this shows another example of transforming programs to make security boundaries more clear. The boundary of concern in this paper was privileged access rights and not privileged data.

Users can be authenticated via rhosts combined with RSA as follows. If the .rhosts file exists in the user's home directory on the remote machine and contains a line containing the name of the client machine and the name of the user on that machine, and if additionally the server can verify the client's host key, only then login is permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing [31].

This method is of particular concern from an information flow security standpoint, as it requires the client to load the private keys of the host in order to verify its identity to the server. Thus, the programmer must be extremely careful to ensure the host's private keys are not leaking either to the server, or an improper location on the local file system.

The analyzed code in this section is from OpenSSH version 3.9, developed by Tatu Ylonen [30]. We omit any portions of the SSH client code that are not significant for the purpose of this example, such as local variable declarations, initialization, and option processing. We also omit details where a simple explanation suffices. These explanations are of the form of special comments: ***/* omitted code */***. Other comments are included for readability.

We now discuss the identification and refactoring of the program, and address the more difficult issues of library calls and IO channels in section 4.

3.1 Identification

The relevant SSH client source code is given in figures 5 and 6. Identification uses the forward slice of the program starting at the input point of the private keys (figure 6 line 7), The slice spans the methods `main`, `ssh_login`, and `ssh_userauth1`, all of which reside in separate files. Only the portions of these methods related to rhosts-RSA authentication are identified. Note that we only load one key in this simplified code, whereas the actual SSH client may load several private keys.

The functions in figure 6 are all almost entirely in the high slice, as they are affected by the private key input. Notably, the input from the server is not part of the high slice, as slicing only shows the statements directly affected by the criterion. However, since these

```
int main(int ac, char **av)
{ /* setup code */
 if (ssh_connect(host, &hostaddr, options) != 0)
  exit(1);

 PRIV_START;
 fd = open(_PATH_HOST_KEY_FILE, O_RDONLY);
 keys[0] = return key_load_private_rsa1(fd,
  _PATH_HOST_KEY_FILE, "", NULL);
 PRIV_END;

 ssh_login(sensitive, host,
  (struct sockaddr *)&hostaddr, pw);

 key_free(keys[0]);
 keys[0] = NULL;
 xfree(keys);

 exit_status = ssh_session();
 packet_close();
 return exit_status;
}

void ssh_login(&sensitive_data,const char *orighost,
struct sockaddr *hostaddr, struct passwd *pw)
{ /* setup code */
 ssh_kex(host, hostaddr);
 ssh_userauth1(local_user, server_user, host,
  sensitive);
}

void ssh_userauth1(const char *local_user, const char
*server_user, char *host, Sensitive *sensitive)
{
 /* Send username *local_user to server */
 type = packet_read();
 if (type == SSH_SMSG_SUCCESS)
  return; /* Authenticated without password */

 if (sensitive->keys[0] != NULL &&
 sensitive->keys[0]->type == KEY_RSA1 &&
 try_rhosts_rsa_authentication(local_user,
 sensitive->keys[0])) {
  goto success;
 }
 /* other authentication methods */
 success: return;
}
```

Figure 5. SSH Client Code

```
static Key * key_load_private_rsa1(int fd, const char
*filename, const char *passphrase, char **commentp)
{
 buffer_init(&buffer);
 cp = buffer_append_space(&buffer, len);

 read(fd, cp, (size_t) len);

 prv = key_new_private(KEY_RSA1); /* initialize key */
 buffer_get_bignum(&buffer, prv->rsa->n);
 buffer_get_bignum(&buffer, prv->rsa->e);
                              /* Read public key from buffer. */

 /* Rest of the buffer is encrypted. Decrypt it. */

 buffer_get_bignum(&decrypted, prv->rsa->d);
                              /* Read private key. */
 buffer_free(&decrypted);
 close(fd);
 return prv; /* The private key. */
}

static int try_rhosts_rsa_authentication(
 const char *local_user, Key * host_key)
{
 /* Send public key to server for authentication. */
 type = packet_read(); /* Server's response, a challenge. */
 respond_to_rsa_challenge(challenge, host_key->rsa);
 BN_clear_free(challenge); /* Delete challenge. */
 type = packet_read(); /* Server Response */
 if (type == SSH_SMSG_SUCCESS) { return 1; }
 return 0;
}

static void respond_to_rsa_challenge(BIGNUM *
 challenge, RSA * prv)
{
 rsa_private_decrypt(challenge, challenge, prv)
 /* Compute MD5 of decrypted challenge plus session id. */
 BN_bn2bin(challenge, buf + sizeof(buf) - len);
 MD5_Init(&md);
 MD5_Update(&md, buf, 32);
 MD5_Update(&md, session_id, 16);
 MD5_Final(response, &md);
 /* Send the response back to the server. */
}
```

Figure 6. SSH Client Code Continued

inputs involve only the RSA algorithm, they should be in the high component.

3.2 Refactored Components

We now refactor the SSH client into high- security and low-security components. The low component of the refactored SSH client appears in figure 7. In the low component, a highlighted `statement` delineates calls to code in the high component. The `main` method calls `ssh_connect` to initiate a connection with the server. Assuming the connection is successfully established, the program proceeds to authentication. At this point, the program loads the private keys for the client machine. Code for loading the private keys has been refactored to the high component. `ssh_login` is then called, which attempts to authenticate the user, possibly using the private keys, which are passed as argument. Upon successful return of `ssh_login` (assuming the user is authenticated), the private keys are discarded via a call to the high component, and the SSH session begins.

The `ssh_login` method is unchanged by the refactoring, as it is merely a conduit for user authentication. The `ssh_userauth1` method transmits the username to the server for logging in. If the server responds with success (i.e. the user has no password), the method returns. Otherwise, it proceeds through the authentication methods, the first of which is combined rhosts-RSA, which has been moved to the high component. Other authentication methods (e.g. password input) are omitted from this simplified code.

Figure 8 shows the refactored high component, which creates a new class, `Sensitive`. The method `loadPrivateKeys` loads the keys of the client machine. The `PRIV_START` macro is first called to give the program root access for reading the keys, `PRIV_END` will later release this access. The method `key_load_private_rsa1` is called to read the key from the file. The `clearPrivateKeys` method destroys the private keys by freeing the memory and setting the corresponding pointer to `NULL`.

```
int main(int ac, char **av)
{ /* setup code */
 if (ssh_connect(host, &hostaddr, options) != 0)
  exit(1);
 Sensitive sensitive();
 sensitive.loadPrivateKeys();

 ssh_login(sensitive, host,
  (struct sockaddr *)&hostaddr, pw);
 sensitive.clearPrivateKeys();

 exit_status = ssh_session();
 packet_close();
 return exit_status;
}

void ssh_login(Sensitive& sensitive,const char
 *orighost, struct sockaddr *hostaddr,
 struct passwd *pw)
{ /* setup code */
 ssh_kex(host, hostaddr);
 ssh_userauth1(local_user, server_user, host,
  sensitive);
}

void ssh_userauth1(const char *local_user, const char
*server_user, char *host, Sensitive& sensitive)
{
 /* Send username *local_user to server */
 type = packet_read();
 if (type == SSH_SMSG_SUCCESS)
  return; /* Authenticated without password */
 if (sensitive.authenticateWithRSARhosts(local_user))
 {
  goto success;
 }
 /* other authentication methods */
 success: return;
}
```

Figure 7. SSH Client Refactored: Low Component

```
class Sensitive {
public:
 Sensitive() { /* Initialization of data structure. */ }
 void loadPrivateKeys() {
  int fd;
  PRIV_START;
  fd = open(_PATH_HOST_KEY_FILE, O_RDONLY);
  keys[0] = return key_load_private_rsa1(fd,
   _PATH_HOST_KEY_FILE, "", NULL);
  /* The data in keys[0] must be labeled sensitive */
  PRIV_END;
 }
 void clearPrivateKeys() {
  key_free(keys[0]);
  keys[0] = NULL;
  xfree(keys);
 }
 bool authenticateWithRSARhosts(char *local_user) {
  bool success = false;
  if (keys[0] != NULL && keys[0]->type == KEY_RSA1
  &&
  try_rhosts_rsa_authentication(local_user, keys[0]))
   success = true;
  }
  return Declassify(success,high);
 }
private:
 Key **keys;
 int nkeys;

 static Key * key_load_private_rsa1(int fd,
 const char *filename, const char *passphrase,
 char **commentp)
 { /* Same code as in figure 6 */ }

 static int try_rhosts_rsa_authentication(
  const char *local_user, Key * host_key)
 { /* Same code as in figure 6 */ }

 static void respond_to_rsa_challenge(BIGNUM *
  challenge, RSA * prv)
 { /* Same code as in figure 6 */ }
```

Figure 8. SSH Client Refactored: High Component

Note both loadPrivateKeys and clearPrivateKeys were initially blocks of code in main, which have been moved into new public methods. authenticateWithRSARhosts contains the code taken from ssh_userauth1 and so is a public method. The return value of this method indicates success or failure of authentication. This value has its *high* label declassified, since information flow control tracks the leak from the private key to this point. Declassification is intensional and returning success or failure of authentication is deemed "safe," as it leaks very little information about the private key.

The entire key_load_private_rsa1 has been placed into the high component as a private method, since it is only accessed by loadPrivateKeys. try_rhosts_rsa_authentication and respond_to_rsa_challenge are also refactored as private methods. The data structure for holding the keys, a Keys struct, along with the number of keys nkeys are now private fields in the Sensitive class. This prohibits access to the keys from outside the class, except through the public methods.

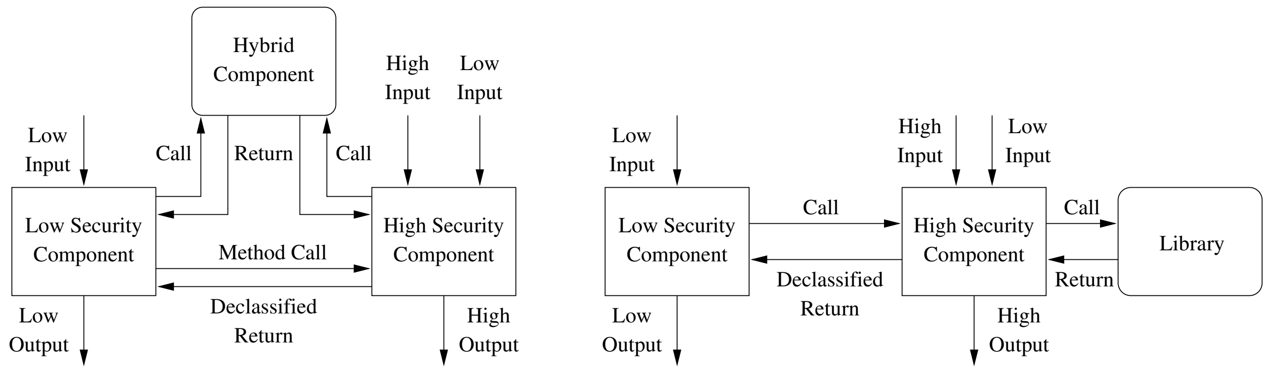
Refactoring moves all of the sensitive code from the initial program to a high component, so that the low code accesses sensitive operations through calls to the public methods in the high component. There are some subtle issues involved in this refactoring, which we discuss in the next section.

4. Challenges of Refactoring Real Programs

In the previous sections, we have described a simple technique for refactoring programs into high and low security components. It would be nice if all programs could quickly and easily be segmented into two portions, being entirely clear which code belongs in each portion, with a simple, concise interface between the two. Unfortunately, for real programs things are not so simple. Real programs use libraries, high and low components may share code, and placement of declassification points is non-trivial. The following sections address these challenges and present techniques to deal with these issues. We illustrate these challenges using the OpenSSH example from the previous section.

4.1 Hybrid Components

Many programs use methods that will be used by both high and low security components. For example, OpenSSH uses buffers for many purposes, both in the high and low components. The act of loading private keys in Figure 6 buffer, which in the end is destroyed using the buffer_free call on line 18. The code for this function is shown in figure 10. This function is called many times in the OpenSSH program, including in the low component (although they do not appear in our shortened example). Since



(a) Program with Hybrid Component

(b) Program with Library Component

Figure 9. Program Refactoring Showing Libraries and Hybrid Components

`buffer_free` is used in both components, it does not belong solely in one component, as it affects both high and low data. Thus, *hybrid* components are necessary for functions that are used for both the high and low security components.

```
void buffer_free(Buffer *buffer)
{
 if (buffer->alloc > 0) {
  memset(buffer->buf, 0, buffer->alloc);
  buffer->alloc = 0;
  xfree(buffer->buf);
 }
}
```

Figure 10. SSH Buffer Code

Program slicing lacks the formalism to describe *label polymorphism*, so we adopt the formalism from information flow type systems, where label types are assigned to expressions that describe the security label of the expression. In order to provide reusable code, methods must be polymorphic in the label types of the arguments and return values, so they can be used on both high and low level data. The need for hybrid components arises due to this label polymorphism.

A hybrid component consists of methods that are label polymorphic and are used in both high and low components. This definition is valid only when the hybrid component has no information flow leaks. In other words, the program is well-typed in an information flow type system. Figure 9(a) illustrates a hybrid component that is accessed by both high and low components.

In the context of adding information flow security to a program that previously had none, what remains is deciding which methods should be in a hybrid component. We address this with the following rules for finding candidates for hybrid methods, and showing which candidates correctly belong in a hybrid component.

- Hybrid methods must actually be used by both high and low components. A method used only by a high component need not be shared—it is more natural to leave it in the high component. Candidates for hybrid methods are identified by intersecting the methods in the high slice with those in the low slice (recall the low slice is the backward slice from a low output point).
- Hybrid methods should not perform any security operations on input data. Thus, hybrid methods should contain no data label, check, or declassify commands. Doing any of these operations clashes with the polymorphic nature of hybrid components.
- Any potential back-channels in a method means it should not be refactored into a hybrid component. For example, static fields create a back-channel for information leakage between two different objects of the same class.

A violation of these rules indicates a method should not be placed in a hybrid component. The first two rules will keep inappropriate code out of the hybrid slice. The last rule is a consequence of the information flow policy: any such back-channel will be flagged as a security violation by the analysis.

Violations of the second rule can be checked by observing any methods identified as hybrid candidates by this process. One must simply make sure that these methods do not contain the security operations. This is an easy syntactic check.

Checking candidate hybrid methods for violations of the third rule is a bit more complex. Sharing data via static fields provides another channel for possible information flow leakage. Hybrid components can accommodate static variables, as long as they reside solely in the high or low component. Information is allowed to flow between two objects in the high component through this channel, which does not violate the information flow policy.

For an example of an unsound static field, suppose the method `buffer_free` was defined in a class `Buff` as in figure 11. Due to the conditional at line 6, the static field `size` may hold the size of the buffer containing the data. This is a security leak if buffers are used in both high and low components, as `Buff` objects in the low component can access high data through the static field `size`.

The high slice will include this leak, thus tainting the code all the way to the low output. Indeed, unless this flow is specifically allowed via declassification, an information flow analysis will show the error. If this use of the static field is desired, the `Buff` class should be put in the high component, and a different class must be used in the low component.

Library and hybrid components are similar, as figure 9 shows. Indeed, both indicate code re-use, although hybrid code is meant only for the current program, whereas library code is usually written by someone other than the programmer, for a more general use.

Figure 9(b) shows a refactored program with a high library component that is used by the high component. Consider the `MD5` calls

```
class Buff {
 static int size;

 void buffer_free(Buffer *buffer)
 {
  if (buffer->alloc > 50) { size = buffer->alloc; }

  if (buffer->alloc > 0) {
   memset(buffer->buf, 0, buffer->alloc);
   buffer->alloc = 0;
   xfree(buffer->buf);
  }
 }
}
```

Figure 11. Buffer Code with Static Field

in lines 41-44 of figure 6. The assumption is that the library methods correctly implement the MD5 hash of the decrypted challenge that is passed to it. In this example, the MD5 library component is used only by the high component; yet it could also be used by the low component, since the library methods do not affect the security level of the arguments.

4.2 Declassification

Placement of declassification points is a central task in refactoring for information flow security. If a program initially has no information flow policy, the refactoring also assists in the initial placement of declassifications. A high slice that crosses the component boundary to the low component signals a leak that should be carefully evaluated and in the end may be deemed "safe", and a declassification statement inserted. Programmers need to be extremely careful when declassifying flows across the component boundary, as one unsound declassification can defeat the security of the whole system. For example, declassifying a password string allows the actual password to leak through a public output channel. The information flow system would not report an error, yet a leakage occurs since the policy was wrong. Placement of declassifications requires significant knowledge of the code, in order to determine when it is "safe" to declassify. Nevertheless, we can help programmers in declassification decisions by establishing some principles.

Principle: Declassification should be done only upon method return. This allows the declassification to be shown in the API. Developers can then see what is being declassified, and determine at a conceptual level whether or not the declassification is warranted, then observe the method's code to make sure the implementation is correct, so no unsafe leaks are made.

```
int rsa_private_decrypt(BIGNUM *out, BIGNUM *in,
 RSA *key)
{ /* Initialization code */
 if ((len = RSA_private_decrypt(ilen, inbuf, outbuf,
  key, RSA_PKCS1_PADDING)) <= 0) {
   error("rsa_private_decrypt() failed");
  } else {
   BN_bin2bn(outbuf, len, out);
  }
 /* Delete buffers */
 Declassify(out,high);
 return len;
}
```

Figure 12. SSH Decryption

The result is a method-based downgrading policy that states which data will be declassified when passed as argument to the method. This principle is related to the downgrading policies of Li and Zdancewic [13], where data is labeled with policies that declare the operations that allow the data to be declassified. For example, a data policy label may state that the data may be declassified if it is passed to the `hash` function.

For a concrete example, consider the `rsa_private_decrypt` method in figure 12 that decrypts `in` and places the result at `out`, which is declassified according to the "safety" of the RSA algorithm not revealing information about the private key. The resulting method-based policy is that `rsa_private_decrypt` removes the label *high* from the output of the method, `out`. An equivalent data label policy could be placed on the private key data at the input point, allowing the data to be declassified by the `rsa_private_decrypt` method. Thus, the private key would carry the downgrading policy.

The result is the same in either case, and the output from `rsa_private_decrypt` will be declassified. The principle behind the similarity is that a safe declassification is a pair consisting of the kind of data, and a function or method on that data; the approach in [13] places the policy on the data, and we place it on the method. From a software engineering perspective, we argue that placing downgrading policies on methods is preferable, since it can be clearly shown in the API for each method, creating a more observable downgrading policy. Data on the other hand gets buried within the code, resulting in buried downgrading policies if the data contains the policies. In addition, the accuracy of declassification is tightly coupled with the method implementation. A change to the method may result in a leak and an unsafe declassification. Policies on data may not be aware of such changes, resulting in failure of the security system. Consider a different implementation of the `rsa_private_decrypt` in figure 13. The highlighted code at line 6 outputs the private key in the case where the decryption failed, leading to a breach of security and an unsafe declassification. If the policies are carried on the data, there is less chance that such a glaring mistake will be caught. By placing downgrading policies on the methods, the programmer is able to inspect the code for correctness of the policy.

```
1:  int rsa_private_decrypt(BIGNUM *out, BIGNUM *in,
2:   RSA *key)
3:  { /* Initialization code */
4:   if ((len = RSA_private_decrypt(ilen, inbuf, outbuf,
5:    key, RSA_PKCS1_PADDING)) <= 0) {
6:     out = key->rsa->d;
7:    } else {
8:     BN_bin2bn(outbuf, len, out);
9:    }
10:  /* Delete buffers */
11:  Declassify(out,high);
12:  return len;
13: }
```

Figure 13. SSH Decryption with Security Hole

Principle: Declassification points should declassify a minimal number of sources, and nearest to the point where the data first becomes watered-down enough to warrant declassification. This will protect against inadvertent declassifications of information, e.g. if a high label comes from two different sources and declassification occurs late, the programmer may miss that one of the values actually is not safe to be declassified. For example, in the `rsa_private_decrypt` in figure 12, the output of the method is declassified, as the programmer determines that this method leaks

```
static int try_rhosts_rsa_authentication(
 const char *local_user, Key * host_key)
{
 /* Send public key to server for authentication. */
 type = highRead(); /* Server's response, a challenge. */
 respond_to_rsa_challenge(challenge, host_key->rsa);
 BN_clear_free(challenge); /* Delete challenge. */
 type = highRead(); /* Server Response */
 if (type == SSH_SMSG_SUCCESS) { return 1; }
 return 0;
}

void ssh_userauth1(const char *local_user, const char
*server_user, char *host, Sensitive *sensitive)
{
 /* Send username *local_user to server */
 type = lowRead();
 /* Same code as in figure 7 */
}

int highRead() {
 return packet_read();
}
int lowRead() {
 return packet_read();
}
```

Figure 14. SSH High and Low Input Channels

a "safe" amount of information. A declassification of this flow that occurs later in the program is poor design, since it occurs far from the point at which the leak is considered "safe."

Violations of the first principle can be captured by an information flow analysis, although it is up to the programmer to determine the proper downgrading policies. The second principle can be enforced by using information flow controls to show how many high security inputs flow into a declassification point. Warnings can be raised so programmers can check this for accuracy. Similarly, a backward slice from a declassification point will show the high inputs that flow to the point, providing another accuracy check.

4.3 Input and Output Channels

Unfortunately, most programs use very few different channels for IO operations, and this sometimes makes it difficult to distinguish between a high channel and a low channel. A good example is the socket connections in the OpenSSH example of section 3. The statement `type = packet_read()` inputs a packet from the socket connection to the server. This occurs in the low component at line 30 of figure 7, where the client is reading the server's response to the username that was just sent. The same statement is used in the high component at line 30 in figure 6 (this code was refactored, but not duplicated in figure 8). This is a reply to the challenge that is decrypted using a private key, leaking a small amount of information. This creates confusion in defining high security output channels, as the underlying connection is the same.

We solve this problem by creating separate high and low channel abstractions, which use the same underlying socket. This results in a clearer design of the program in an abstract sense, as the program can now distinguish between channels.

In figure 14, we show a portion of refactored code that separates the high and low input channels. In the high component, input calls `try_rhosts_rsa_authentication` are replaced by `highRead()`. In the low component, input calls are replaced by `lowRead()` in the `ssh_userauth1` method. We also note that packets are read in the same manner once the client loop has started.

Separation of high and low output channels is similar, although a declassification may be necessary for the underlying channel. Assuming the underlying channel is low, data from the high component must be declassified before being sent. In SSH, data is placed in packets via `packet_put` commands, which serve as the underlying channel. Since the underlying channel is low, a high output channel can be defined as follows.

```
high_packet_put_char(c) {
 packet_put_char(Declassify(c));
}
```

This method should then be used in the high component wherever a high output is desired.

Channel separation provides a better abstract model of the security policy of the program; however, security holes may still arise due to declassifications of high data on the channel. Control of channel separation necessitates the use of temporal properties to specify when the channel is high and low, and that accesses to the channel are correct. For example, in SSH once the authentication phase is over, high data should not be sent over the socket connection. A flow-sensitive analysis is needed to control temporal properties of shared channels.

Debug channels and log files are common sources of security leaks, as they are usually less heavily scrutinized than other code. Refactored programs must separate such channels so that debug information does not end up in an insecure file location. A high slice will show if information is leaking from the high component into a low debug channel, which can be fixed during refactoring.

5. Related Work

As stated in the introduction, other researchers have observed that programs may be refactored into high and low security parts. Li and Zdancewic [13] mention this as an avenue for proving noninterference, and additionally note that their *relaxed noninterference* can be formalized by allowing some functions to depend on high inputs, which are treated as downgrading policies.

Amtoft and Banerjee [2] make a similar observation, along with showing how forward slicing can help. They show how slicing can be used to eliminate all assignments that depend on a high variable `h`, as well as all conditionals and loops with tests that depend on `h`. The result is a program where every high command is replaced by `skip`. Refactoring of the high security code is not discussed, and most of the issues addressed in this paper do not arise, since the language they use is small.

Privilege separation involves partitioning a program into two separate processes: a privileged monitor that executes any privileged operations and an unprivileged slave that does the rest. The goal is to reduce the amount of code that runs in privileged mode, reducing the likelihood of attackers gaining superuser access. Provos et al. gave a manually privilege-separated version of OpenSSH [18]. Brumley and Song developed Privtrans [5], an automated program for privilege separation. They use a static dataflow analysis to discover which function calls should be executed by the privileged monitor, then transform the source code by rewriting these calls with wrapper functions. This analysis only accounts for direct flows, where privileged data is directly passed to a function, or assigned to a variable. Our approach aims to also account for indirect flows, to produce a stronger information flow security guarantee. Their design focuses solely on process separation, whereas we actually move code to separate components. This act of physically separating the components makes the security policy more clear which will increase reliability.

Program slicing was first addressed by Weiser [29], and has been an active area of research for many years; see [25] for an overview. Secure information flow has been around even longer, beginning with early work by Bell and LaPadula [4] and Denning [7]; this area has also been a well-studied research topic [20]. However, it has not been until fairly recently that the commonalities between the two schools have been examined.

Abadi et al. defined the Dependency Core Calculus (DCC) [1], and showed how the SLam calculus [12], a typed lambda calculus with information flow annotations, and a slicing calculus could both be encoded in DCC. This allows noninterference to be applied to program slicing, although they use a type system for computing slices as opposed to the usual dependency graph approach.

Snelting et al. showed how backward slices satisfy a noninterference criterion [22]. This differs from Abadi et al. [1], since they assume slices are computed using a system dependence graph (SDG); noninterference is shown for the Goguen and Meseguer definition [9]. They later describe how to use path conditions in program dependence graphs (PDGs) for information flow control [11]. The slicing approach has an advantage over the type-based approach of being flow-sensitive, but it is less rigorously defined than a type-based approach.

None of the above works discuss the benefits and challenges of refactoring programs for information flow security purposes. Similarly, refactoring and restructuring techniques have been well-studied [14], yet the emphasis is not on information flow security. Rather, refactoring generally emphasizes design issues, such as version compatibility, reducing redundancies, etc. The idea of using program slices to guide refactorings has popped up before [27], but not in an information flow context.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*. To appear.

[3] Anindya Banerjee and David Naumann. Using access control for secure information flow in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003., 2003.

[4] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, 1975.

[5] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[6] Frank W. Calliss. Problems with automatic restructurers. *SIGPLAN Notices*, 23(3):13–21, 1988.

[7] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[8] Mike Downen. Find out what's new with code access security in the .net framework 2.0. `http://msdn.microsoft.com/msdnmag/issues/05/11/CodeAccessSecurity/`, November 2005.

[9] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[10] GrammaTech, Inc. Codesurfer. `http://www.grammatech.com/products/codesurfer/index.html`.

[11] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[12] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[13] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[14] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[15] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[16] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[17] Franois Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[18] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[19] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. Indus. `http://indus.projects.cis.ksu.edu/`.

[20] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Jounal on Selected Areas in Communications*, 21(1), January 2003.

[21] Scott F. Smith and Mark Thober. Securing data at Java IO boundaries. `http://www.cs.jhu.edu/~mthober/securingdata06.pdf`. Draft.

[22] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*. To appear.

[23] Sun Microsystems, Inc. Security code guidelines. `http://java.sun.com/security/seccodeguide.html`, February 2000.

[24] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[25] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[26] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Eng.*, 8(1):89–120, 2001.

[27] Mathieu Verbaere. Program slicing for refactoring, 2003. MSc thesis, University of Oxford.

[28] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[29] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[30] Tatu Ylonen. Ssh client program, 1995.

[31] Tatu Ylonen. Ssh manpage, Nov. 8 1995.