Implicit Flows: Can't Live With 'Em, Can't Live Without 'Em

Dave King[1], Boniface Hicks[2], Michael Hicks[3], and Trent Jaeger[1]

[1] The Pennsylvania State University
{dhking,tjaeger}@cse.psu.edu
[2] Saint Vincent College
fatherboniface@acm.org
[3] University of Maryland, College Park
mwh@cs.umd.edu

Abstract. Verifying that programs trusted to enforce security actually do so is a practical concern for programmers and administrators. However, there is a disconnect between the kinds of tools that have been successfully applied to real software systems (such as taint mode in Perl and Ruby), and information-flow compilers that enforce a variant of the stronger security property of noninterference. Tools that have been successfully used to find security violations have focused on *explicit flows* of information, where high-security information is directly leaked to output. Analysis tools that enforce noninterference also prevent *implicit flows* of information, where high-security information can be inferred from a program's flow of control. However, these tools have seen little use in practice, despite the stronger guarantees that they provide.

To better understand why, this paper experimentally investigates the explicit and implicit flows identified by the standard algorithm for establishing noninterference. When applied to implementations of authentication and cryptographic functions, the standard algorithm discovers many real implicit flows of information, but also reports an extremely high number of false alarms, most of which are due to conservative handling of unchecked exceptions (e.g., null pointer exceptions). After a careful analysis of all sources of true and false alarms, due to both implicit and explicit flows, the paper concludes with some ideas to improve the false alarm rate, toward making stronger security analysis more practical.

1 Introduction

The last decade has seen a proliferation of static analysis tools that analyze commodity software to discover potential security vulnerabilities. For example, tools have been developed, in research and industry, to uncover SQL injection vulnerabilities [16, 9], missed access control checks [25], user-kernel pointer bugs [13], and format string vulnerabilities [6, 21]. At their core, these tools are quite similar in that they track the flow of security-relevant data through the program. For example, many programs receive untrusted input from the filesystem or the network. If such data is unduly trusted it can be used for malicious purposes,

e.g., to construct an unexpected SQL query string or `printf` format string that leaks secrets or compromises the program. An analysis may track the flow of input data to ensure it is santized or verified before it is trusted. An analysis may dually check whether secret data, such as a cryptographic key or password, may flow to a public channel; if so, this constitutes an information leak [3].

The last decade has also seen the development of security-typed programming languages [20] (such as Jif [18], a variant of Java) that enforce flavors of the security property of noninterference [11, 19]. Like the tools mentioned above, such languages ensure that no untrusted data *explicitly* flows to trusted contexts (and dually, that no secret data flows to public contexts). However, unlike these tools, security-typed languages also ensure that there are no illegal *implicit* flows, which arise due to control-dependent assignments. For example, given two boolean-typed variables `H` and `L`, the code fragments `L := H` and `if (H) then L := true; else L := false` are semantically equivalent. In the first, `H` explicitly flows to `L` via an assignment. In the second, no direct assignment takes place, but nevertheless `H` has the same value as `L` at the conclusion of execution; thus we say that `H` *implicitly* flows to `L`.

To our knowledge, implicit flow checking is not performed by mainstream tools. On the upside, tracking implicit flows could reveal more security vulnerabilities. On the downside, an analysis that tracks implicit flows could waste developer time, either by being too inefficient or producing too many false alarms (warnings that do not correspond to actual problems). A natural question is "do the benefits of implicit flow analysis outweigh the costs?"

This paper presents the results of a study we performed toward answering this question. We analyzed several libraries for information leaks using JLift, an interprocedural extension of the Jif security-typed language [14] and carefully analyzed the results to see how often JLift alarms real implicit information flows, and how often they were false alarms. The code we analyzed implemented security-critical functions: three different authentication methods in the J2SSH Java SSH library and three different cryptographic libraries from the Java implementation of the Bouncy Castle Cryptography API. We chose these applications for two reasons. First, they are security-critical, so understanding their information leaks is important. Second, password checking and encryption directly manipulate sensitive data, and are known to induce illegal flows implicitly. Therefore, the prevalence of true leaks in these routines suggests the best case for finding potential leaks due to implicit flows, while the prevalence of false alarms points to the human costs of analyzing an implicit flow checker's results when doing so is likely to be worthwhile.

To perform the analysis, we labeled sensitive data, such as passwords or cryptographic keys, as *secret*, and labeled output channels as *public*, so that JLift emits an alarm each time it discovers secret information could be inferred over a public channel. For our benchmarks, implicit flows caused 98% (870 out of 887) of the alarms, and of the 162 alarms identifying true information leaks, 145 of these (89%) were due to implicit flows. On the other hand, there was a tremendous number of false alarms (725), all of which were due to implicit flows (i.e., 83% of

the implicit flow alarms, or 725 out of 870, could not arise at runtime). Examining these further we find that the overwhelming majority (845 out of 870) of implicit flow alarms arise from the potential to throw an exception after examining sensitive data. Moreover, while the false alarm rate for flows induced by normal conditional statements is 30%, the rate of false alarms for exception-induced flows is much higher, at 85%. The false alarm rate from *unchecked* exceptions, such as `NullPointerException`s and `ArrayIndexOutOfBoundsException`s, is higher still: 757 of the 845 exception-induced flows were due to unchecked exceptions, and 706 of these (or 93.2%) were false alarms.

We draw two conclusions from these results. On the one hand, implicit flow checking can be valuable: JLift emitted 145 alarms that correspond to true implicit flows of secret information. On the other hand, the human cost of dealing with the many false alarms is likely to be quite high, and could well be the reason that most tools perform no implicit flow checking. However, because the high rate of false alarms comes from a few identifiable sources (notably exceptions, particularly unchecked exceptions), we see much potential for improvement. In particular, we suggest that implicit flow checking tools: (1) employ more powerful, secondary analyses for ruling out spurious alarms due to infeasible unchecked exceptions; (2) rank the errors reported, beginning with explicit flows, followed by implicit flows not from exceptions, and finally reporting implicit flows due to exceptions; (3) employ annotations so that programmers can indicate code that has been checked by an alternative method, such as a human code review or extensive testing.

2 Program Analysis for Security

A large body of work has explored the use of programming languages and analyses, both static and dynamic, to detect security vulnerabilities. Many analyses aim to discover *source-sink* properties, or *explicit flows*. For example, to discover a possible format string vulnerability, a programmer could use the CQUAL [10] static analyzer to give `printf` the type `printf(untainted char* fmt, ...)`. This type specifies that `printf`'s first argument must not come from an untrusted source. Input functions are then labeled as producing potentially tainted data, e.g., the type `tainted char* getenv(char *name)` indicates that the value returned from `getenv` is possibly tainted, since it could be controlled by an attacker. Given this specification, CQUAL can analyze a source program to discover whether data from a source with type `tainted char*` is able to explicitly flow, via a series of assignments or function calls, to a sink of type `untainted char*`. If such a flow is possible, it represents a potential format string vulnerability.

Explicit flow analysis tools [9, 16, 24] can be used to detect a variety of integrity violations, such as SQL injection vulnerabilities [16, 9], format string vulnerabilities [6, 21], missed access control checks [25], and user-kernel pointer bugs [13]. These tools can also be used to check confidentiality-oriented properties, to ensure that secret data is not inadvertently leaked. In this case, the programmer would label sensitive data, such as a private key, as `secret` and ar-

guments to output functions as `public`, and the analyzer would ensure that secret data never flows to public channels (except under acceptable circumstances, e.g., after an access control check). Scrash [3] uses CQUAL in this way to identify sensitive information that should be redacted from program crash reports.

While common, explicit flows are but one way in which an attacker is able to gain information about a program's execution for malicious purposes. The other possibility is to use an *implicit flow* by which, in the case of confidentiality properties, the attacker learns secret information indirectly, via a control channel. As a simple illustration, consider the following program:

```
secret int x;
void check(public int y) {
  if (x > y) then printf("greater\n");
  else printf("smaller\n");
}
```

CQUAL would not report a violation for this program: the secret integer `x` is not leaked via an explicit flow to the public console. However, *information* about `x` is leaked, in particular whether or not `x` is greater than the attacker-controlled variable `y`. Invoked repeatedly, with different values of `y`, such a function could ultimately leak all of `x`. It has been shown that cryptosystem implementations can end up leaking the private key if they contain certain kinds of implicit flows, e.g., by reporting the stage in which a decryption failed [1, 2, 23]. Conversely, an attacker's input can influence a program to corrupt trusted data, e.g., by turning a `setuid`-invoking program into a confused deputy [5].

These problems are violations of the more general information flow security property of noninterference [11]. For confidentiality, a program that enjoys noninterference will not leak secret information to public outputs in any manner, whether via implicit or explicit flows. *Security-typed programming languages* [20] such as Jif [18] enforce noninterference via type checking. Program types are annotated with security labels like the `secret` and `public` qualifier annotations above, and security correctness is guaranteed through type checking: type-correct programs do not leak secret information on public channels. Thus, using a security-typed language, the `check` function above would be correctly flagged as a potential leak of information.

3 Security Type Checking and Implicit Flows

The question we address in this paper is whether implicit flow checking as done by security-typed languages is efficacious—is the signal worth the noise? This section describes the standard security-type checking algorithm used to detect implicit flows and considers possible sources of imprecision. The next section examines the number and sources of true and false alarms when using this algorithm to analyze some security-critical Java libraries.

In a security-typed language, types are annotated with *security labels*, analogous to type qualifiers: the integer type **int** can be labeled with the label `high`

```
1 int{Public} authenticate(AuthenticationServer auth,
2                           SshAuthRequest msg) {
3     Key{Secret} key = reader.readBinaryString();
4     if (checkKey == 0) {
5         if (verify.acceptKey(msg.getUsername(), key)) {
6             SshAuthPKOK reply = new SshAuthPKOK(key);
7             auth.sendMessage(reply);
8             return READY; }
9         else return FAILED; } }
```

Fig. 1. Example Authentication Code From J2SSH

as **int**{`high`} to signify a high-secrecy integer. The labels are ordered, inducing a subtyping relation $\preceq$ on labeled types. In particular, because `low` is less secret than `high`, **int**{`low`} is a subtype of **int**{`high`}, written **int**{`low`} $\preceq$ **int**{`high`}. If `h` and `l` are respectively high and low secrecy variables, then the assignment statement `h := l` is allowed, as **int**{`low`} $\preceq$ **int**{`high`}. On the other hand, `l := h` is *not* allowed, preventing an illegal explicit flow, as **int**{`high`} $\not\preceq$ **int**{`low`},

The statement **if** `h == 0` **then** `l := 1` **else** `l := 0` contains an implicit flow of information from `h` to `l`, as by observing `l` we learn information about the value of `h`. The standard way [20] to check for implicit flows is to maintain a security label PC_ℓ for the *program counter*. This label contains the information revealed by knowing which statement in the program is being executed at the current time. PC_ℓ is determined by the labels of guards of any branches taken: if the program branches on a condition that examines a high-security value, PC_ℓ is set to `high` while checking the code in the branches. When a variable with label m is assigned, we require $\mathrm{PC}_\ell \preceq m$. In the above statement, the branch condition (`h == 0`) causes the PC_ℓ to be `high` when checking the branches, meaning that any assignment to `l` is illegal and so the code is rejected.

For a more realistic example, consider the code in Figure 1, taken from the J2SSH implementation of public-key authentication. We use the label `Secret` to represent `high` security, and `Public` to represent `low` security. This code is executed when a user is attempting to authenticate using an SSH key. The `acceptKey` method checks if the specified key is an accepted key for the user that is attempting to log in. As this contains information about the system, `acceptKey` returns a `Secret` boolean value, causing PC_ℓ to become `Secret`. As such, the value `READY` returned from the branch must also be considered `Secret`, but notice that we have annotated the return value of `authenticate` as **int**{`Public`}, so this constitutes an implicit information leak.

While sound, the use of PC_ℓ to find implicit flows is conservative. Consider the program **if** `h == 0` **then** `l := 1` **else** `l := 1`. This program leaks no information—the value of `l` is always 1, no matter what the value of `h`—but the type system rejects it as an illegal flow. A more pernicious source of false alarms is the throwing of exceptions. For example, the Java code `obj.equals(otherObj)` throws a `NullPointerException` at run time if `obj` is **null**. If such an exception

occurs while PC_ℓ is `high`, the resulting termination of the program (assuming the exception is not caught) is publicly-visible, and thus is an implicit flow of information. In the example in Figure 1, such an exception thrown within the call to `sendMessage` could reveal whether the given key was accepted. Therefore, if the type system cannot prove that `obj` is always non-**null**, then dereferencing it must be considered as possibly throwing a null pointer exception. An analogous situation arises with other exceptions, such as array bounds violations and class cast exceptions.

Jif's type checker makes some attempt to prove that certain objects cannot be **null** and that some array accesses are legal, but its analysis is fairly simplistic. For example, the analysis operates only on local variables (not fields or formals), and is only intraprocedural. Programmers are thus forced to work within the limits of the analysis to limit false alarms. For example, it is a common practice to copy fields and formal method arguments into local variables to check if the objects are **null** before invoking methods on them. Redundant null checks and empty **try**/**catch** blocks (where the programmer presumes no exception can actually be thrown but the analysis cannot detect this) are also common.

While Jif's analysis could be improved, determining whether a runtime exception could occur or not is undecidable [15], so no analysis can be perfect. To show the general difficulty of the problem, consider the following code (taken from the public key authentication routine in J2SSH):

```
1 byte[] decode(byte[] source, int off, int len) {
2       int len34 = len * 3 / 4;
3       byte[] outBuff = new byte[len34];
4       ...
5 }
```

In this code, the statement **new byte**[`len34`] could throw a `NegativeArraySizeException`, meaning that the program attempted to create a byte array with a negative size (if `len34` is negative). As this code is invoked after checking whether a username is valid on the system, the program crashing here after a negatively-sized array is created could reveal this information to the requester. While the passed-in value `len` is meant to always be positive, there is no easy way, in general, for a program analysis to determine this. In our manual analysis, we determined that the only call to `decode` by the public key authentication method was with the variable `bytes.length`, where `bytes` was an array returned by `String.getBytes`(), a quantity that will always be non-negative.

4 Experiments

To determine the impact of implicit program flows in analyzing real code, we analyzed six implementations of security functions: (1) three different authentication methods in the J2SSH Java SSH library:[4] and (2) three different cryp-

[4] J2SSH is available from `http://sourceforge.net/projects/sshtools`.

tographic libraries from the Java implementation of the Bouncy Castle Cryptography API.[5] The J2SSH authentication methods that we investigated were password-based, keyboard-interactive, and public-key-based. The Bouncy Castle implementations that we investigated were RSA Encryption (using SHA1 Digests), MD5 Hashing, and DES Encryption. To gather statistics about the existing runtime exceptions in these codebases, we used JLift, an interprocedural extension of the Jif security-typed language [14].

The results from our investigations are summarized in Figure 2. Our experiments show the following:

- Implicit flows corresponded to most of the leaks of information in these applications. In the SSH authentication methods, there were no explicit information leaks, while there were only a handful of explicit leaks in the cryptography libraries. In total, leaks corresponding to explicit flows made up less than 2% of the reported alarms, and none of the explicit flows corresponded to a false positive.
- The implicit flows arising from unchecked runtime exceptions dominate all other flows. Specifically, 757 out of 870 (87 %) of alarms caused by implicit flows were due to the five types of runtime exceptions (Null Pointer, Array Out of Bounds, Class Cast, Negative Array, Arithmetic). These results are summarized in Figure 4.
- Most flows due to unchecked exceptions were caused by program paths that could not be executed at runtime. Specifically, we manually verified that 706 alarms (out of 824 total exceptional alarms) could not occur at run time.

4.1 Methodology

We chose to analyze two different codebases for information leaks: an SSH server and a cryptographic library. These programs are security-critical, and thus constitute a worthy target of security analysis. While it would be unsurprising to find implicit (and explicit) flows in these programs since they manipulate secret information in an observable way (e.g., by outputting the result of a password check or emitting ciphertext), our interest is of effectiveness: does the standard algorithm, when applied to mature code written in a natural style, identify the true explicit and implicit information flows without producing a deluge of false alarms?

4.2 Analysis Methodology

Details of Security Analysis. To perform an information-flow analysis without explicitly converting the code into Jif as well as investigating how a implicit flow checker behaved when applied to code written in a natural style, we used JLift [14], a modification of the Jif compiler to perform a whole-program

[5] BouncyCastle implementations for Java and C# are available at `http://www.bouncycastle.org/`.

information-flow analysis on Java programs. From a set of seed labels, JLift propagates security information across procedure calls by determining a set of summary constraints for each method, a standard method in static analysis [22]. JLift relies on the Jif compiler to generate these constraints: in particular, it annotates types with a security label and uses a program counter to detect implicit flows. We ran JLift using a context-sensitive treatment of methods, associating a unique set of summary constraints to each instance of a method call. Our experience has been that context sensitivity is required for an accurate information-flow analysis [14].

Annotation of JLift. To analyze these programs for security errors, we labeled protected data as high security and analyzed the resulting flows. As JLift does not report all possible program paths that lead to a violation, we ran the tool multiple times, suppressing earlier warnings by adding some "declassification" annotations (which indicate that certain flows should be ignored). To efficiently catalogue exceptions that could be thrown at runtime that would affect the program counter, we first used the analysis to determine which methods could be invoked from a high program counter location. We then had JLift enumerate the runtime exceptions, determined by an interprocedural propagation analysis, that could possibly be thrown by those methods. We investigated each of these runtime exceptions to determine if they could be thrown at run time.

Following Jif's behavior, we did not directly analyze the program flows that could occur when invoking a library function at runtime. For conservativity, we assigned each instance of a class defined in a library a label variable L and treated each value passed to or returned from methods or fields as having label L. This prevented secure information from being laundered by being passed through library functions.

False Alarms. We define a *false alarm* as an alarm that the analysis tool reports that corresponds to an infeasible program path. For example, if the variable `v` is always not **null**, then an alarm raised by the statement `v.foo()` attributed to a `NullPointerException` at this point would be false. We marked an alarm as true when we could manually confirm the flagged path could actually execute at run time; all other alarms were considered false. As an example, Jif's null pointer analysis is only performed on local variables; all fields are considered to be possibly null. As such, many times a field dereference was incorrectly flagged as problematic even though all feasible paths initialized the field (e.g., in the constructor) before dereferencing it.

Several places in the code explicitly throw an exception: for example, if a block to encrypt was too large for the RSA cipher, the program would throw a `DataLengthException`. We did not count explicitly thrown exceptions or exceptions from library calls (for example, `IOException` on file open) as false alarms. The code for MD5 hashing explicitly threw exceptions at only three locations, as opposed to DES, which threw exceptions at nineteen locations: this accounts for some of the difference in false positive rate between the two.

Program	Security Function	# Alarms			False Alarms		False Alarm
		Total	Explicit	Implicit	Explicit	Implicit	Rate
J2SSH	Password	7	0	7	0	3	42.86 %
	Keyboard Interactive	23	0	23	0	19	82.61 %
	Public Key	170	0	170	0	111	65.29 %
BouncyCastle	RSA	218	3	215	0	186	86.51 %
	MD5	209	4	205	0	199	97.07 %
	DES	260	10	250	0	207	82.80 %

Fig. 2. False alarm rates for the three authentication methods present in J2SSH. The first and column columns give the application and security function that was analyzed. The third column gives the total number of alarms, with the fourth and fifth columns containing the number of alarms raised by both potential explicit and implicit flows. The sixth and seventh column gives the number of false alarms (subdivided between explicit and implicit flows), while the seventh column gives the overall percentage of false alarms among all of the alarms reported by JLift.

4.3 J2SSH

J2SSH is a Java SSH library that contains an SSH server implementation that has three different authentication methods: password, keyboard-interactive, and public-key authentication. This code most directly leaks information to potential attackers. If the programmer is not careful, information leaks in an authentication system risk giving away important information about the system, such as legal user names or even passwords (perhaps subject to a brute-force attack).

J2SSH handles authentication by allowing the host operating system to provide a native authentication mechanism. This object represents the operating system mechanism for checking and changing passwords. Our experiment was thus to see how the J2SSH server code protected the native authentication mechanism. We marked information associated with the native authentication mechanism as `Secret` and assumed that the authentication method returned a `Public` result.

There were only three means of communicate with the user in the J2SSH server: (1) the returned result from authentication (a **boolean**), (2) messages sent by the server back to the client, and (3) abnormal termination of the SSH server. There were no explicit flows of information in these codebases, as each of the authentication methods behaved similarly to the code from Figure 1: based on the result of a secret query, a value is returned to the user, a message is sent, or the program terminates.

4.4 Bouncy Castle Cryptographic Implementations

We also ran our analysis on the implementations of three cryptographic implementations from the Java implementations of the Bouncy Castle Cryptographic APIs: RSA asymmetric-key encryption (with SHA1 digests), MD5 hashing, and DES symmetric-key encryption.

```
1  int newKey[32], boolean pc1m[56], boolean pcr[56];
2  for (int j = 0; j < 56; j++) {
3      int l = pc1[j];
4      pc1m[j] = (key[l >>> 3] & bytebit[l & 7]) != 0; }
5  for (int i = 0; i < 16; i++) {
6      int l, m, n;
7      if (encrypting) { m = i << 1; } else { m = 15 - i << 1; }
8      n = m + 1; newKey[m] = (newKey[n] = 0);
9      for (int j = 0; j < 28; j++) {
10         l = j + totrot[i];
11         if (l < 28) { pcr[j] = pc1m[l]; } else { pcr[j] = pc1m[l - 28]; }
12     } for (int j = 28; j < 56; j++) {
13         l = j + totrot[i];
14         if (l < 56) { pcr[j] = pc1m[l]; } else { pcr[j] = pc1m[l - 28]; }
15     } for (int j = 0; j < 24; j++) {
16         if (pcr[pc2[j]]) { newKey[m] |= bigbyte[j]; }
17         if (pcr[pc2[j + 24]]) { newKey[n] |= bigbyte[j]; } } }
```

Fig. 3. Code from the DES key scheduling code, where the original secret key is split into two keys to operate on half of a 64-bit encryption string. Nearly every line of the above code was flagged as possibly throwing an exception.

For these codebases, we labeled both the keys and the data being encrypted or hashed as secret. For each function, the code performing the encryption contained many low-level bitwise operations, including accesses to pre-defined constant arrays and bit-shifts. This lead to many more possible array access errors reported in the Bouncy Castle cryptographic libraries than in J2SSH. This also accounts for the slightly higher false positive rate in the Bouncy Castle cryptographic implementations as compared to J2SSH.

Figure 3 shows a typical example of code where it is difficult to rule out impossible runtime exceptions. The code was taken from the implementation of DES and contained a very large false alarm rate: nearly every line of the algorithm was flagged as possibly throwing at least one exception. Automatically verifying that this code cannot throw an exception is difficult: it requires knowledge of the contents custom-built arrays `bytebit`, `totrot`, `bigbyte`, knowing that these arrays cannot change and carefully maintaining knowledge of possible bounds on each variable. With some effort, we were able to manually verify that each line of the above code could not throw a runtime exception.

4.5 Discussion

Figure 4 contains statistics about implicit flows, exceptional flows, and alarms raised by the five most commonly occurring runtime exceptions. From this figure, we can see that most alarms were caused by implicit flows, and that most of these implicit flows were caused by exceptions. Of these, five exceptions make up the bulk of the false alarms reported by the Jif program analysis: null pointer errors,

Exception	# Alarms	False Alarms	False Alarm Rate	% of Total Alarms
all flows	887	725	81.74 %	100 %
all implicit	870	725	81.74 %	98.08 %
all exceptions	824	706	79.59 %	92.90 %
null pointer	475	455	95.79 %	53.78 %
array bounds	233	204	87.55 %	26.27 %
non-exceptional implicit flows	46	14	30.43 %	5.19 %
class cast	24	22	91.67 %	2.78 %
negative array	20	20	100.0 %	2.76 %
arithmetic exception	5	5	100.0 %	0.56 %

Fig. 4. Rates of false alarms per exception for the most commonly-occurring runtime exceptions. The first column gives the type of exception, the second gives the number of alarms, the third gives the number of those that were due to an code path that is not realizable at runtime, and the fourth gives the rate of false alarms. The fifth column shows the percentage of total alarms (number of alarms from this category divided by total alarms) that were caused by this specific exception type.

array bounds errors, class cast errors, negative array errors, and arithmetic errors (divide by zero). The applications that we looked at had no explicit flows that corresponded to infeasible runtime paths; this may be due to the small number of explicit flows that these applications contained.

Our experimental results suggest that developers analyzing code for information-flow security should follow an iterative process: first analyze explicit flows, then analyze implicit flows that occur because of non-exceptional program paths, and finally analyze the remaining implicit flows.

Most of the implicit flows reported by the analysis that were not false alarms corresponded to technical violations of noninterference. For example, in DES, the eventual size of the output buffer for ciphertext (publicly available) depended on the size of the data to be encrypted. There were also several authentication flows that revealed information about whether a username was valid or not, but these occurred after the password was successfully verified, making these flows difficult to exploit. For example, when a login attempt was valid but the user was required to change his or her password, the server would send a message with this information.

A handful (3) of implicit flows that we marked as false alarms were not caused by the type-based nature of the analysis. These alarms occurred because the J2SSH KBI authentication method assigned three variables to both high security and low security values and the Jif program analysis treated all occurrences of those variables as high security data.

We make no judgment about the value of true alarms. Each of the true alarms represented a violation of noninterference. Methods for determining the severity of these alarms (for example, statically determining for each alarm a quantitative

upper bound of number of bits leaked, as in recent work [17]) are beyond the scope of this paper.

One may wonder how our results with JLift speak to Jif programming as it is done today, as Jif has been used to build several substantial systems, including JPMail, a mail client [12] and a Civitas, remote voting system [7]. The answer is simple: the programmer must work around the imprecision of various analyses in order get the program to type check. As mentioned before, programmers often copy fields into local variables because the null pointer analysis in the compiler is only done on local variables. Empty catch statements, signaling the programmer's belief that a runtime exception should not affect the security of the program counter, are also quite common. To quantify these observations, we examined the Civitas code base, which contains over 13000 lines of Jif Code. We found 568 empty catch statements to handle unexpected runtime exceptions. Of these, 429 of these caught exceptions were given the variable name `imposs`, indicating that the programmers believed that it was impossible for these errors to occur. Of the remainder, 13 of them were given the variable name `unlikely`, indicating that the programmers thought it was not likely that these errors would occur. While programmer belief does not guarantee all of these exceptions are impossible or unlikely, these coding constructions speak to the need for a sound but less burdensome way for developers to handle implicit flows arising from runtime exceptions.

5 Towards Painfree Noninterference

In this section we suggest several approaches for handling the high rate of false alarms.

Modern programming languages support robust features for error handling and recovery. As an imperative object-oriented language, Java gives more control over errors than previous languages like C and C++: if an error occurs in code, the Java runtime can catch and handle it, rather than simply terminating. However, the possibility for failure at every method call or array access in Java makes it difficult to perform an accurate information-flow analysis. Once the program counter has been raised to secret by a high-security conditional or thrown exception, even one that might itself be a false alarm, a sound static analysis must treat the execution of the rest of the program as conditional, meaning that every possible exception raised after this must be reported and handled.

From Figure 4, it is clear that better program analyses would aid in reducing the number of false alarms. Currently, most null pointer accesses cannot be proven safe by Jif, and as almost any line of Java code can throw a `NullPointerException`, this leads to an unacceptably high number of false alarms. Improved analyses could reduce this false alarm rate, but due to the fundamental undecidability of determining if a null pointer access is safe [15] combined with the difficulty of maintaining invariants for every line of code (as seen in Figure 3), these analyses will likely remain imperfect.

One way to improve the power of these analyses is to rely more on programmer annotations. For example, a programmer could annotate when an object could possibly be null, rather than the opposite: this would shift the burden of dealing with null pointer analyses to annotating which variables could possibly store the null value. Security-typed languages could also be integrated with tools such as ESC/Java [4, 8] that can verify these annotations and so prove properties of variables and array bounds while reducing the number of false positives reported by the security analysis. Adding a new block primitive to the Jif programming language that indicated that the enclosed code could not terminate abnormally would allow Jif programmers to make trusted sections of code more explicit without relying on empty **try/catch** blocks, as in the Civitas codebase.

As we have seen, a sound security analyses must therefore consider a large number of runtime program paths, not all of which can occur at runtime. Flow Caml [19] is a security-typed language based on Caml Lite, a member of the ML family of functional programming languages. As ML does not have the concept of a null pointer, Flow Caml does not require a null pointer analysis for programmers to use the language. It may be possible to, by changing the semantics of the underlying Java language, modify the kinds of flows that can occur at runtime and so reduce the number of false positives reported by a static analysis. Another possibility is to change our security model to insert a global system declassifier so that certain classes of implicit flows (those with a high noise rate) are allowed to leak information about secret data. If this is adopted, care must be taken to ensure that these flows cannot be repeatedly exploited to gain a substantial amount of information from the system.

Finally, different programming models could aid in reducing the number of false alarms. If a section of code can be encapsulated as only taking input and returning an output (rather than possibly terminating in the middle of its computation), then any implicit flows that occur during this computation can be folded into the input (through a **try/catch** block, for example). If every operation that is performed inside of a high program counter region will not throw an exception, then the only implicit flows that can occur during a program are updates to the system's global state performed by these operations (such as message sends and receives). It may be possible to automate this approach.

6 Conclusion

In this paper, we have outlined the impact of considering *implicit flows* in a security analysis. Our experiments show that when checking mature, security-critical code, the standard type-based algorithm reports a strikingly large number of implicit flows as false alarms. Most of these implicit flows are induced by unchecked exceptions that a static analysis is not able to prove are infeasible at runtime. If we are to make security analysis with implicit flows practical, better tools and different programming techniques are necessary.

References

1. Black, J., and Urtubia, H. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *Proceedings of the 11th USENIX Security Symposium* (2002).
2. Bleichenbacher, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *CRYPTO* (1998), pp. 1–12.
3. Broadwell, P., Harren, M., and Sastry, N. Scrash: a system for generating secure crash information. In *Proceedings of the 12th conference on USENIX Security Symposium* (2003).
4. Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. Springer-Verlag, 2006, pp. 342–363.
5. Chen, H., Wagner, D., and Dean, D. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 171–190.
6. Chen, K., and Wagner, D. Large-scale analysis of format string vulnerabilities in Debian Linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security* (2007).
7. Clarkson, M. R., Chong, S., and Myers, A. C. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy* (2008), pp. 354–368.
8. Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *PLDI* (June 2002), vol. 37, pp. 234–245.
9. Fortify Software. Fortify. `http://www.fortify.com/`.
10. Foster, J. S., Fähndrich, M., and Aiken, A. A theory of type qualifiers. In *PLDI* (1999), pp. 192–203.
11. Goguen, J. A., and Meseguer, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.
12. Hicks, B., Ahmadizadeh, K., and McDaniel, P. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)* (Miami, FL, December 11-15 2006).
13. Johnson, R., and Wagner, D. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 9–9.
14. King, D., Jaeger, T., Jha, S., and Seshia, S. A. Effective blame for information-flow violations. In *FSE 2008* (November 2008). To appear.
15. Landi, W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 323–337.
16. Martin, M., Livshits, B., and Lam, M. S. Finding application errors and security flaws using PQL: a program query language. In *OOPLSA* (New York, NY, USA, 2005), ACM, pp. 365–383.
17. McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. In *PLDI* (2008), pp. 193–205.
18. Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL* (January 1999), pp. 228–241.
19. Pottier, F., and Simonet, V. Information flow inference for ML. In *POPL* (New York, NY, USA, 2002), ACM Press, pp. 319–330.

20. Sabelfeld, A., and Myers, A. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003).
21. Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium* (2001).
22. Sharir, M., and Pnueli, A. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications* (1981), Prentice Hall, pp. 189–234.
23. Vaudenay, S. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (London, UK, 2002), Springer-Verlag, pp. 534–546.
24. Xie, Y., and Aiken, A. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems 29*, 3 (2007).
25. Zhang, X., Edwards, A., and Jaeger, T. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 33–48.