Lazy Programs Leak Secrets

Pablo Buiras and Alejandro Russo

Dept. of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
{buiras,russo@chalmers.se}

Abstract. To preserve confidentiality, information-flow control (IFC) restricts how untrusted code handles secret data. While promising, IFC systems are not perfect; they can still leak sensitive information via covert channels. In this work, we describe a novel exploit of *lazy evaluation* to reveal secrets in IFC systems. Specifically, we show that lazy evaluation might transport information through the *internal timing covert channel*, a channel present in systems with concurrency and shared resources. We illustrate our claim with an attack for *LIO*, a concurrent IFC system for Haskell. We propose a countermeasure based on restricting the implicit sharing caused by lazy evaluation.

1 Introduction

Information-flow control (IFC) permits untrusted code to safely operate on secret data. By tracking how data is disseminated inside programs, IFC can avoid leaking secrets into public channels—a policy known as non-interference [4]. Despite being promising, IFC systems are not flawless; the presence of covert channels allows attackers to still leak sensitive information.

Covert channels arise when programming language features are misused to leak information [6]. The tolerance to such channels is determined by their bandwidth and how easy it is to exploit them. For instance, the termination covert channel, which exploits divergence of programs, has a different bandwidth in systems with intermediate outputs than in batch processes [1].

Lazy evaluation is the default evaluation strategy of the purely functional programming language Haskell. This evaluation strategy has two distinctive features which can be used together to reveal secrets. Firstly, since it is a form of *non-strict evaluation*, it delays the evaluation of function/constructor arguments and let-bound identifiers until their denoted values are needed. Secondly, when the evaluation of such expressions is required, their resulting value is stored (cached) for subsequent uses of the same expression, a feature known as *sharing* or *memoisation*. This is known as *call-by-need* semantics or simply lazy evaluation. In Haskell, a *thunk*, also known as a delayed computation, is a parameterless closure created to prevent the evaluation of an expression until it is required at a later time. The process of evaluating a thunk is known as *forcing*. While lazy evaluation does not affect the denotation of expressions with respect to non-strict semantics, it affects the timing behaviour of programs. For instance, if a function argument is used more than once in the body of a function, it is almost always faster to use lazy evaluation as opposed to call-by-name, since it avoids re-evaluating every occurrence of the argument.

From a security point of view, it is unclear what type of semantics (non-strict versus strict) is desirable in order to deal with covert channels. In sequential settings, Sabelfeld and Sands [10] suggest that a non-strict semantics might be intrinsically safer than a strict one. This observation is based on the ability to exploit the *termination covert channel*. Although it could avoid termination leaks, lazy evaluation can compromise security in other ways. For instance, Rafsson et al. [9] describe how to exploit the Java (lazy) class initialisation process to reveal secrets. Not surprinsingly, lazy evaluation might also reveal secrets through the *external timing covert channel*. This channel involves externally measuring the time used to complete operations that may depend on secret data.

More interestingly, and totally unexplored until this work, lazy evaluation might transport information through the *internal timing covert channel*. This covert channel arises by the mere presence of concurrency and shared resources. Malicious code can exploit it by setting up threads to race for a public shared resource and, depending on the secret, affecting their timing behaviour to determine the winner. With lazy evaluation in place, thunks become shared resources and forcing their evaluation corresponds to affecting the threads' timing behaviour—subsequent evaluations of previously forced thunks take practically no time.

We present an attack for *LIO* [12], a concurrent IFC system for Haskell, that leverages lazy evaluation to leak secrets. *LIO* presents countermeasures for internal timing leaks based on programming language level abstractions. Since *LIO* is embedded in Haskell as a library, lazy evaluation, as a feature that primarily affects pure values, is handled by the host language. Lazy evaluation is essentially built into Haskell's internals, hence there are no programming language-level mechanisms for inspecting or creating thunks that could be used to implement a countermeasure. Thunks for pure values are transparently injected into *LIO* computations, so the library could not be capable of explicitly considering whether they have been memoised at any given time.

This paper is organised as follows. Section 2 briefly recaps the basics of *LIO*. Section 3 presents the attack. Section 4 describes a possible countermeasure. Conclusions are drawn in Section 5.

2 *LIO*: a concurrent IFC system for Haskell

In purely functional languages, computations with side-effects are encoded as values of abstract data types called monads [8]. In Haskell, there are monads for performing inputs and outputs (monad *IO*), handling errors (monad *Error*), etc. The IFC system *LIO* is simply another monad in which security checks are performed before side-effects are performed.

The *LIO* monad keeps track of a *current label*. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labelled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Since the current label protects all the variables in scope, in practical programs we need a way of manipulating differently-labelled data without monotonically increasing the current label. For this purpose, *LIO* provides explicit references to labelled, immutable data through a parametric data type called *Labeled*. A locally accessible symbol can bind, for example, a value of type *Labeled l Int* (for some label type l), which contains an *Int* protected by a label different from the current one. Function *unlabel* :: *Labeled l a* $\rightarrow$ *a*[1] brings the labelled value into the current lexical scope and updates the current label accordingly.

LIO also includes IFC-aware versions of well-established synchronisation primitives known as *MVar*s [5]. A value of type *LMVar* is a mutable location that is either empty or contains a value. Function *putLMVar* fills the *LMVar* with a value if it is empty and blocks otherwise. Dually, *readLMVar* empties an *LMVar* if it is full and blocks otherwise.

3 A lazy attack for *LIO*

Figure 1 shows the attack for *LIO*. The code essentially implements an internal timing attack [11] which leverages lazy evaluation to affect the timing behaviour of threads. We assume the classic two-point lattice (of type *LH*) where security levels L and H denote public and secret data, respectively, and the only disallowed flow is the one from H to L. Function *attack* takes a public, shared *LMVar lmv*, and a labelled boolean *secret* (encoded as an integer for simplicity). The goal of *attack* is to return a public integer equal to *secret*, thus exposing an *LIO* vulnerability. In isolation, all the threads are secure. When executed concurrently, however, *secret* gets leaked into *lmv*. For simplicity, we use *threadDelay n*, which causes a thread to sleep for n micro seconds, to exploit the race to *lmv*—if such an operation was not allowed, using a loop would work equally well.

The attack proceeds as follows. Threads A and B do not start running until thread C finishes. This effect can be easily achieved by adjusting the parameter *delay_C*. The role of thread C is to force the evaluation of the list *thunk* when the value of *secret* is not zero ($s \not\equiv 0$). To that end, function *traverse* goes over *thunk*, returning one of its elements. Condition $n > 0$ always holds and it is only used to force Haskell to fully evaluate the closure returned by *traverse*. Threads A and B will eventually start racing. Thread A executes the command *traverse thunk* before writing the constant 1 into *lmv* (*putLMVar lmv* 1). Thread B delays writing 0 into *lmv* (*putLMVar lmv* 0) by some (carefully chosen) time *delay_B*.

[1] Symbol :: introduces type declarations and $\rightarrow$ denotes function types.

```
attack :: LMVar LH Int → Labeled LH Int → LIO LH Int
attack lmv secret
  = do let thunk = [1 .. constant] :: [Int]
         -- Thread C
       forkLIO (do s ← unlabel secret
                   when (s ≢ 0) (do n ← traverse thunk
                                    when (n > 0) (return ())))
       threadDelay delay_C
         -- Thread A
       forkLIO (do n ← traverse thunk
                   when (n > 0) (putLMVar lmv 1))
         -- Thread B
       forkLIO (do threadDelay delay_B
                   putLMVar lmv 0)
       w ← takeLMVar lmv
       _ ← takeLMVar lmv
       return w
```

Fig. 1: Attack exploiting lazy evaluation

If $s \not\equiv 0$, *thunk* will have already been evaluated when thread A traverses its elements, thus taking less time than thread B's delay. As a result, value 1 is first written into *lmv*. Otherwise, thread B's delay is shorter than the time taken by thread A to force the evaluation of *thunk*. In this case, value 0 is first written into *lmv*. Variable w observes the first written value in *lmv*, which will coincide with the value of the secret. The precise values of parameters *constant*, *delay_C*, and *delay_B* are machine-specific and experimentally determined.

The following code shows the magnification of the attack for a list of secret integers.

```
magnify :: [Labeled LH Int] → LIO LH [Int]
magnify ss = do lmv ← newEmptyLMVar L
                mapM (attack lmv) ss
```

Function *magnify* takes a list of secret values *ss* (of type [*Labeled LH Int*]). The magnification proceeds by creating the public *LMVar* (*newEmptyLMVar L*) needed by the attack. Function *mapM* sequentially applies function *attack lmv* (i.e. the attack) to every element in *ss* and collects the results in a public list ([*Int*]).

Below, we present the final component required for the attack:

```
traverse :: [a] → LIO LH a
traverse xs = return (last xs)
```

This function simply returns the last element of the list given as argument.

The code for the attack can be downloaded from `http://www.cse.chalmers.se/~buiras/LazyAttack.tar.gz`.

4 Restricting sharing

We propose a countermeasure based on restricting the sharing feature of lazy evaluation. Specifically, we propose duplicating shared thunks when spawning new threads. In that manner, sharing gets restricted to the lexical scope of each thread. Thunks being forced in one thread will then not affect the timing behaviour of the others. To illustrate this point, consider the shared *thunk* from Figure 1. If this countermeasure was implemented, forcing the evaluation of *thunk* by thread C would not affect the time taken by thread A to evaluate *traverse thunk*, making the attack no longer possible. An important drawback of this approach is that there would be a performance penalty incurred by disabling sharing among threads. Benchmarking and evaluation would be necessary to determine the full extent of the overhead inherent in the technique. Presumably, programmers could restructure their programs to minimise the effect of this penalty.

As an optimisation, it is possible to only duplicate thunks denoting pure expressions. Thunks denoting side-effecting expressions can be shared across threads without jeopardising security. The reason for that relies on *LIO*'s ability to monitor side-effects. If a thread that depends on the secret forces the evaluation of side-effecting computations, the resulting side-effects are required to agree with the IFC policy. For instance, threads with secrets in lexical scope can only force thunks that perform no public side-effects; otherwise *LIO* will abort the execution in order to preserve confidentiality.

To implement our approach, we propose using **deepDup**, an operation introduced by Joachim Breitner [2] to prevent sharing in Haskell. Essentially, **deepDup** takes a variable as its argument and creates a private copy of the whole heap reachable from it, effectively duplicating the argument thunk and disabling sharing between it and the original thunk. In his paper, Breitner shows how to extend Launchbury's natural semantics for lazy evaluation [7] with **deepDup**. The natural semantics is given by a relation $\Gamma : t \Downarrow \Delta : v$, which represents the fact that from the heap Γ we can reduce term t to the value v, producing a new heap Δ. It is the relation between Γ and Δ which captures heap modifications caused by memoisation. In this setting, the rule for **deepDup** is

$$\frac{\begin{array}{c}\Gamma, x \mapsto e, x' \mapsto \hat{e}[y'_1/y_1 \ldots, y'_n/y_n], (y'_i \mapsto \mathbf{deepDup}\ y_i)_{i \in 1\ldots n} : x' \Downarrow \Delta : z \\ \mathrm{ufv}(e) = \{y_1, \ldots, y_n\} \qquad x', y'_1, \ldots, y'_n \text{ fresh}\end{array}}{\Gamma, x \mapsto e : \mathbf{deepDup}\ x \Downarrow \Delta : z}$$

where $\mathrm{ufv}(e)$ is the set of unguarded[2] free variables of e and $\hat{e}$ is e with all bound variables renamed to fresh variables in order to avoid variable capture when ap-

[2] Function $\mathrm{ufv}(e)$ is defined as the set of free variables that are not already marked for duplication, i.e. $\mathrm{ufv}(\mathbf{deepDup}\ x) = \emptyset$, and in the rest of the cases it is inductively defined as usual.

plying substitutions. Note that **deepDup** x duplicates all the thunks reachable from x in a lazy manner: the free variables $y_1, \ldots, y_n$ are replaced with calls to **deepDup** for each variable, so these duplications will not be performed until those variables are actually evaluated. Laziness is necessary to properly handle cyclic data structures, since the duplication process would loop indefinitely if it were to eagerly copy all thunks for such structures. As explained below, this design decision has important consequences for security.

In practice, we would use this primitive every time we fork a new thread: we take the body of the new thread m_1 and the body of the parent thread m_2, and replace them with **deepDup** m_1 and **deepDup** m_2. Due to the lazy nature of the duplication performed by **deepDup**, it is necessary to duplicate both thunks, i.e., m_1 and m_2. Consider two threads A and B with current labels L and H, respectively, and suppose that they both have a pointer to a certain thunk x in the same scope. If we only duplicated the thunk in A (the public thread), thread B could evaluate parts of x depending on the secret, before they have been duplicated in thread A—recall that **deepDup** is lazy. This would cause the evaluation of the same parts of the duplicated version of x in A to go faster, thus conveying some information about the secret to thread A. In addition, note that it is not possible to determine in advance—at the time *forkLIO* is called—which thread will raise its current label to H. Therefore, we must take care to duplicate all further references to shared thunks every time a fork occurs.

As a possible optimisation, we advise designing a data dependency analysis capable of over-approximating which expressions are shared among threads. Once the list of expressions (and their scope) has been calculated, we would proceed to instrument the code, introducing instructions that duplicate only the truly shared thunks at runtime, as opposed to duplicating every pure thunk in the body of each thread. We believe that HERMIT [3] is an appropriate tool to deploy such instrumentation as a code-to-code transformation.

5 Conclusions

We describe and implement a new way of leveraging lazy evaluation to leak secrets in *LIO*, a concurrent IFC system in Haskell. Beyond *LIO*, the attack points out a subtlety of IFC for programming languages with lazy semantics and concurrency. We propose a countermeasure based on duplicating thunks at the time of forking in order to restrict sharing among threads. For that, we propose to use the experimental Haskell package *ghc-dup*. This package provides operations that copy thunks in a lazy manner. Although convenient for preserving program semantics, such design decision has implications for security. To deal with that, our solution requires duplicating thunks for both the newly spawned thread and its parent. As future work, we will implement the proposed countermeasure, prove soundness (non-interference), evaluate its applicability through different case studies, and introduce some optimisations to reduce the amount of duplicated thunks.

Acknowledgments We would like to thank Andrei Sabelfeld, David Sands, and the anonymous reviewers for useful comments. Special thanks to Edward Z. Yang, who mentioned the work by Joachim Breitner to us. This work was funded by the Swedish research agency VR, STINT, and the Barbro Osher foundation.

Bibliography

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the European Symp. on Research in Computer Security (ESORICS)*. Springer-Verlag, 2008.

[2] J. Breitner. dup – Explicit un-sharing in Haskell. *CoRR*, abs/1207.2017, 2012.

[3] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2012.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[5] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1996.

[6] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[7] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1993.

[8] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[9] W. Rafnsson, K. Nakata, and A. Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), Jan. 2013.

[10] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1), Mar. 2001.

[11] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.

[12] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2012.