Securing Compilation Against Memory Probing

Frédéric Besson, Alexandre Dang, Thomas Jensen

Securing Compilation Against Memory Probing

Frédéric Besson
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
frederic.besson@inria.fr

Alexandre Dang
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
alexandre.dang@inria.fr

Thomas Jensen
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
thomas.jensen@irisa.fr

ABSTRACT

A common security recommendation is to reduce the in-memory lifetime of secret values, in order to reduce the risk that an attacker can obtain secret data by probing memory. To mitigate this risk, secret values can be overwritten, at source level, after their last use. The problem we address here is how to ensure that a compiler preserve these mitigation efforts and thus that secret values are not easier to obtain at assembly level than at source level. We propose a formal definition of Information Flow Preserving program Transformations in which we model the information leak of a program using the notion of Attacker Knowledge. Program transformations are validated by relating the knowledge of the attacker before and after the transformation. We consider two classic compiler passes (Dead Store Elimination and Register Allocation) and show how to validate and, if needed, modify these transformations in order to be information flow preserving.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Software security engineering**; *Logic and verification*;

KEYWORDS

Secure Compilation; Side-channels; Dead Store Elimination; Register Allocation; Information-flow Preservation

ACM Reference Format:
Frédéric Besson, Alexandre Dang, and Thomas Jensen. 2018. Securing Compilation Against Memory Probing. In *The 13th Workshop on Programming Languages and Analysis for Security (PLAS'18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3264820.3264822

1 INTRODUCTION

Writing secure code is notoriously hard. It gets almost impossible if aggressive compiler optimisations may turn a secure source program into an insecure binary code. This can happen because of compiler bugs [22] but also due to optimisations which exploit undefined behaviours of languages like C to generate code that does not match the expectation of the programmer [21].

Using formal methods (*e.g.*, abstract interpretation [3] or deductive verification [12]), it is possible to prove the absence of undefined behaviours of the source code. To avoid compiler bugs, verified compilers, *e.g* CompCert [13] or CakeML [20], ensure that the observational behaviour of the source code is preserved by the compilation process.

However, security is a non-functional property that is not captured by the standard semantics of programs and therefore may not be preserved even if the program has a defined semantics and the compiler is correct. Recently, D'Silva *et al.* put focus on the "Correctness-Security Gap in Compiler optimizations" [7]. Similarly, the CERT at CMU recommends to "beware of compiler optimizations"[1] and provides the code example in Figure 1 to illustrate the problem. Here, the programmer limits the lifetime of a sensitive variable by overwriting its value using a memset. The semantics of

```
1 void getPassword(void) {
2   char pwd[64];
3   if (GetPassword(pwd, sizeof(pwd))) {
4     /* Checking of password, secure operations,
      etc. */
5   }
6   memset(pwd, 0, sizeof(pwd));}
```

Figure 1: Non-compliant erasure

C states that *An actual implementation needs not evaluate [...] an expression if it can deduce that its value is not used [...]*[11, 5.1.2.3]. As a result, a compliant C compiler is allowed to remove the call to memset and break the security of the code. The advisory also lists so-called compliant code with the caveat that *the programmer [should] inspect the generated assembly code in the optimized release build to ensure that memory is actually cleared [...].* This is not very satisfactory.

Recently, Yang *et al.* [23] have empirically studied the effectiveness of counter-measures and propose a generic memset which adapts to the platform to provide the maximum security. They also propose a secured version of *Dead Store Elimination*, the optimisation responsible for removing the memset. Simon *et al.* [19] take another stance at the problem and propose to insert a compiler pass which secures the code by zeroing stack frames on context switches. These works provide in-depth empirical studies but do not formalise the end-to-end security property they intend to preserve or enforce.

In this work, we propose a formal definition of preservation of information leakage which, if verified by program transformations, ensures that the target code is not less secure than the source code, with respect to a passive but strong attacker able to read an arbitrary

[1] CERT MSC06-C

PLAS'18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5993-1/18/10…$15.00
https://doi.org/10.1145/3264820.3264822

amount of memory. We stress that our purpose is not to enforce a security property; neither at the source level, nor at the target level. What we seek to identify is a general property that provides a formal account of how optimisations (such as the elimination of dead stores as in Figure 1) increase the information leakage and therefore render the optimised program insecure.

Our attacker model (ability to read arbitrary memory) is quite strong and one may wonder whether compiler optimisations preserve the information leakage. In this paper, we review some optimisations and show that some of them need to be adapted. Our contributions can be summarised as:

- We propose a notion of preservation of information leaks and assess its relevance against a list of simple transformations.
- We show how to strengthen our initial proposal to capture information leaks due to partial observations.
- We present sufficient conditions for the absence of information leaks which provide convenient reasoning principles.
- We review two optimisations, dead store elimination and register allocation; and show how they can be adapted to preserve information leakage.

The rest of the paper is organised as follows. In Section 2, we propose a simple definition of preservation based on the notion of Attacker Knowledge. In Section 3, we present a strengthened definition which tracks finer information flows due to partial observations and is our definitive proposal of an Information Flow Preserving (IFP) transformation. Section 4 presents sufficient conditions for an IFP transformation which are easier to reason about in practice. Section 5 shows how the previous reasoning principles apply to two standard program optimisations: dead store elimination and register allocation. We discuss some extension of our formal model in Section 6. Related work is presented in Section 7 and Section 8 concludes.

2 INFORMATION-FLOW PRESERVATION

In this section, we present our formal definition of preservation of information leakage. We define our attacker model and propose a first version of what an Information Flow Preserving Transformation is. We evaluate this proposal against a series of program transformations, explain how it validates (or invalidates) simple transformations.

2.A Rationale for our Attacker Model

Our attacker model formalises the reason why *reducing the lifetime of sensitive data* is a good security practice. Basically, the risk to mitigate is that an attacker might be able to obtain sensitive data by reading the memory. For unsafe languages, *e.g.* C, any attacker code running in the same memory space can access arbitrary location and obtain sensitive data. Even if only trusted code is running, sensitive data can still be leaked through security vulnerabilities such as buffer overflows. In any case, to reduce the scope of these attacks, it makes sense to reduce the lifetime of sensitive data. Our attacker model is also relevant for type-safe languages, *e.g.* Java, where the memory is managed and reclaimed by a garbage collector. In this case, it is not possible for a source code program to read chunks of memory that is not accessible. Hence, if we consider an attacker being another type-safe program linked against the sensitive program, zeroing sensitive data would not be a relevant counter-measure. However, it still makes sense to protect against an attacker with physical access to the memory and therefore able to scan the memory content. In this context, it is worth overwriting sensitive data long before the occurrence of memory collection. Our attacker model covers both cases.

We consider an attacker which has total knowledge of the source and target code and can read arbitrary amount of memory. The attacker has a perfect knowledge of the memory layout and can target precisely the location of secret information. We will define what it means for a program transformation to be secure with respect to this attacker. Intuitively, we intend to ensure that an attacker does not get an advantage after the program transformation. In other words, the memory of the target program does not leak more information than the source code. In the following part, we formalise a simple but representative setting to illustrate these ideas.

2.B Formal setting

For simplicity, we consider deterministic, terminating executions and make the assumption that the attacker has only access to the final memory. As we explain in Section 6, these restrictions can be relaxed. In particular, it seems a reasonable trade-off for attackers to have memory access at the end of each function call.

2.B.1 Knowledge of the Attacker. We consider a finite, bit-addressable memory

$$\mathit{Memory} = \mathit{Addr} \to \mathbb{B}$$

where $\mathbb{B} = \{0, 1\}$ is the set of boolean values. A program $p \in \mathit{Program}$ runs from an initial configuration $c \in \mathit{Config}$ and produces a final memory $m \in \mathit{Memory}$. This is written $(p, c) \Downarrow m$. For the sake of the presentation, we only consider terminating programs *i.e.*, $\forall c. \exists m. (p, c) \Downarrow m$. We discuss in Section 6 how to lift this restriction. In order to model the information leak of an execution, we adapt the standard information-flow notion of Attacker Knowledge [1] to our simple setting.

Definition 2.1 (Attacker Knowledge). Given a program $p \in \mathit{Program}$ and a memory $m \in \mathit{Memory}$, the knowledge of the attacker ($\mathcal{K}(p, m)$) is the set of initial configurations leading to the observation of the memory m. Formally, we have

$$\mathcal{K}(p, m) = \{c \in \mathit{Config} \mid (p, c) \Downarrow m\}$$

The notion of *Attacker Knowledge* characterises the uncertainty of the attacker about the initial configurations of the program. For a program p and a memory m, consider the following extreme cases. If $\mathcal{K}(p, m) = \mathit{Config}$, the attacker has no information about the initial configuration because to her all of them compute the same memory m. On the contrary, if $\mathcal{K}(p, m) = \{c\}$ for some c, the attacker has perfect knowledge: she knows for sure that the program p is run from the configuration c. Hence, from a security standpoint, a bigger set $\mathcal{K}(p, m)$ is better.

2.B.2 Information Flow Preserving Transformation as Increase of Attacker Knowledge . Intuitively, a program transformation is secure against our attacker if her knowledge does not increase by examining the memory after the program transformation. Definition 2.2 formally defines a partial order on programs such that $p \preccurlyeq p'$ is read as *the program p' leaks no more information than*

Original program
f: x=0; **return** 0;
Dead store elimination
g: skip; **return** 0; ✗
Overwrite with constant value
h: x=1; **return** 0; ✓
Overwrite with leaked value
i: x=y; **return** 0; ✓
Leaking through direct flow
j: a=x; x=0; **return** 0; ✗
Overwrite direct flow
k: a=x; x=0; a=0; **return** 0; ✓
Leaking through indirect flow
l: {**if**(x) a=0 **else** a=1}; x=0; **return** 0; ✗

Figure 2: Transformed programs

program p. For convenience, we say that p leaks more information than p', keeping in mind that the relation is reflexive.

Definition 2.2 (Security Ordering). A program p leaks more information than a program p' (written $p \preccurlyeq p'$) if their respective Attacker Knowledge are ordered such that

$$\forall c, m, m'. \quad \left.\begin{array}{c}(p,c) \Downarrow m \\ \wedge \\ (p',c) \Downarrow m'\end{array}\right\} \Rightarrow \mathcal{K}(p,m) \subseteq \mathcal{K}(p',m')$$

Based on Definition 2.2, we are ready to define what it means for a program transformation to be information-flow preserving (see Definition 2.3).

Definition 2.3. A program transformation $T \in \mathit{Program} \to \mathit{Program}$ is Information Flow Preserving if and only if we have

$$\forall p, (p \preccurlyeq T(p))$$

Definition 2.3 ensures that after a transformation the knowledge set of the attacker is getting bigger and therefore that she has less knowledge about the initial configurations leading to her observation. To simplify the definitions, we only consider source-to-source transformations. To capture transformation between source and target language, the previous definitions can be generalised in a straightforward way to distinguish between the source and target semantics.

2.C Reality check

Definition 2.3 is elegant but it should also faithfully capture an intuitive notion of security *i.e.*, reject program transformations deemed insecure and accept program transformations deemed secure. In the following, we consider the simple Program f of Figure 2 and several transformed versions of it. The Program f takes as input configuration the values of the variables x and y; zeroes the variable x and returns 0. As all the transformed programs compute the same value, *i.e.*, 0, these are perfectly fine transformations from a compiler correctness point of view. In Figure 2, we mark with a ✓ the programs verifying Definition 2.2 and with a ✗ those which do not. The results are explained in details in Example 2.4.

Example 2.4. Consider Program g obtained by removing the assignment to the variable x from Program f. From a compiler point of view, this is a correct transformation because the value of x does not contribute to the result. As our motivating example shows (see Figure 1), removing *dead store* leaks more information to the attacker and should therefore be ruled out. Therefore we challenge our Definition 2.3 with the transformation from f to g which is insecure. Figure 3 exhibits this example.

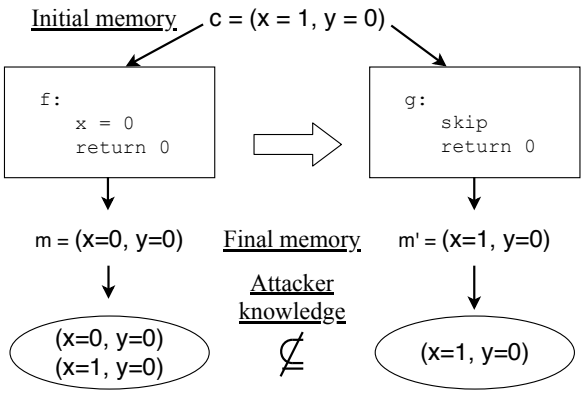

Figure 3: Counter-example to Knowledge inclusion

Consider the initial configuration $c = (x = 1, y = 0)$. The attacker observes the memory $m = (x = 0, y = 0)$ for Program f and the memory $m' = (x = 1, y = 0)$ for Program g. For Program f, because the value of the variable x is overwritten, the attacker has no information about its initial value. Therefore, the Attacker Knowledge is

$$\mathcal{K}(f, m) = \{(x = 0, y = 0); (x = 1, y = 0)\}$$

For Program g, the value of x is not overwritten and therefore the attacker has perfect knowledge of the initial configuration.

$$\mathcal{K}(g, m') = \{(x = 1, y = 0)\}$$

As a result, for configuration c, we have

$$\mathcal{K}(f, m) \not\subseteq \mathcal{K}(g, m')$$

As a result, Definition 2.3 does not hold, therefore, as expected, the transformation from f to g is not information-flow preserving.

For the other examples of Figure 2, we give less formal explanation of whether they are information-flow preserving according to Definition 2.3. Consider Program h and Program i which both overwrite the value of variable x. Instead of the value 0, h uses the constant 1 and i uses the value of variable y. In terms of Attacker Knowledge, this difference vanishes as both assignments have the effect of erasing any information about the initial value of variable x. Furthermore, in both programs, the value of y is leaked, therefore, leaking it twice in Program i does not change the Attacker Knowledge. For any observed memory m, m' or m'', we have

$$\mathcal{K}(f, m) = \mathcal{K}(h, m') = \mathcal{K}(i, m'')$$

and the property holds for the Program h and Program i.

Program j introduces a direct information-flow from the initial value of variable x to variable a. It overwrites x but not a. As the

variable a contains the exact same information as the initial variable of x, Program j is rejected. Program k also introduces the same direct information-flow but overwrites both copies *i.e.* a and x. In this case, the property holds. Program l has the same semantics as Program j but through an indirect information-flow instead of a direct information-flow. As the Attacker Knowledge is a semantics definition, it is immune to syntactic changes and therefore Program l is rejected for the exact same reason as Program j.

3 INFORMATION FLOW PRESERVATION

Definition 2.3 has the advantage of simplicity and matches our expectation for the programs of Figure 2. Nonetheless, as shown by the counter-example discussed in Section 3.A, it is still too permissive. In this section, we present (Definition 3.4), a refinement of Definition 2.3, which considers a hierarchy of attackers performing partial observations.

3.A Partial Observation and Information Leaks

Consider the programs f and f' of Figure 4. Formally, the transformation verifies Definition 2.3. Yet, the transformed program f' is intuitively easier to attack. In the following, we explain why an attacker with full observational power does not capture partial information leaks and show how to strengthen Definition 2.3 (see Definition 3.4 of Section 3.D).

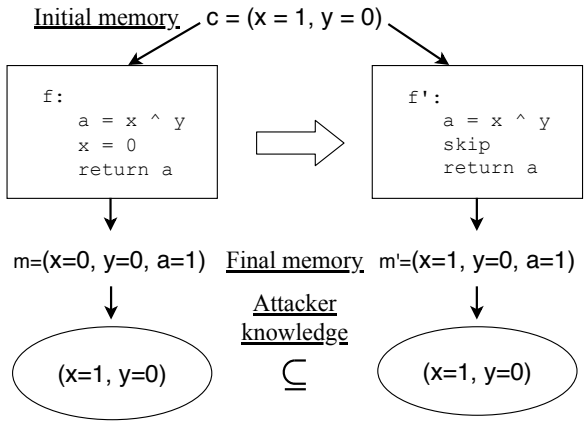

Figure 4: Paradox: not erasing encryption keys preserves information leakage

The intent of program f is best explained using a cryptographic analogy. The program f encrypts the message y using the one-time pad key x by performing an exclusive or with the message. Then, to make it hard for an attacker to recover the key, the program f overwrites the key x. The program f' performs the same computation but does not overwrite the key. Intuitively, the transformation reduces security as the key x is leaked.

However, the transformation verifies Definition 2.3. Consider the scenario of Figure 4 where both programs run from the initial configuration $c = (x = 1, y = 0)$. For Program f', the program computes the memory $m' = (x = 1, y = 0, a = 1)$. Because the value of x is not overwritten, the attacker gains perfect knowledge of the initial configuration c: $\mathcal{K}(f', m') = \{c\}$. For Program f, the value of x is erased and an attacker cannot directly observe its value. However, an attacker can still deduce the value of x from the values of a and y by solving the equation $1 = x \wedge 0$. As the only solution is such that $x = 1$, we have: $\mathcal{K}(f, m) = \{c\}$. A similar reasoning can be made with any initial configuration and therefore the transformation satisfies Definition 2.3. Yet, this goes against our intuition that overwriting variable x makes the program more secure and should therefore be preserved by program transformations. In the following, we propose a strengthened security property which rules out the previous transformation.

3.B Discussion

In order to refine Definition 2.3, we first discuss why zeroing the key does not result in a less secure program. A first argument is that, for Program f', the key is present in-the-clear in the final memory whereas extracting it from Program f requires some computation. Following this path would require to equip the attacker with computational resources and relate the effort required to extract some piece of information. Another argument is that, for Program f, deducing 1 bit of key requires 2 bits of observation (both the plain-text and the cipher-text) whereas, for Program f', this can be done by observing directly 1 bit of the key. As a result, after the transformation, an attacker with a weaker observational power can reconstruct the key. In the following, we formalise this second approach which has the advantage of being purely logical and therefore model attackers without any computational limit.

3.C Partial Attacker Observation

We will represent attacker observations as the set of partial functions $Addr \rightharpoonup \mathbb{B}$ ordered by the usual pointwise ordering:

$$o \sqsubseteq o' \quad \textit{iff} \quad \forall a \in dom(o), a \in dom(o') \wedge o(a) = o'(a)$$

We generalise this ordering to total functions. Hence, for an observation o and a memory m, $o \sqsubseteq m$ reads *o is a partial observation of m.*

Definition 3.1 (Attacker Observation). Given a memory m and a number n of bits, we write $obs(m, n)$ for the set of partial observations of the memory m limited to n bits. Formally, we have:

$$obs(m, n) = \{o \in Addr \rightharpoonup \mathbb{B} \mid card(dom(o)) = n \wedge o \sqsubseteq m\}$$

Definition 3.2 generalises the notion of Attacker Knowledge for an observation.

Definition 3.2 (Attacker Knowledge from Observation). For a given number of bits n and a memory m, the attacker knowledge for a given observation $o \in obs(n, m)$ is defined by

$$\mathcal{K}_n^m(p, o) = \bigcup_{o \sqsubseteq m'} \mathcal{K}(p, m')$$

If the observation is total ($n = card(Addr)$), we get the $obs(m, n) = \{m\}$ and the only memory such that $m \sqsubseteq m'$ is m itself. Under those conditions, Definition 3.2 coincides with Definition 2.1.

$$\mathcal{K}_n^m(p, o) = \mathcal{K}(p, m)$$

In the general case, as the observation o is partial, all the memories $\bar{m}$ such that $o \sqsubseteq \bar{m}$ are indistinguishable to the attacker and are

potential results of the program. As the result, the Attacker Knowledge obtained from the observation o is the union of the knowledge obtained for each potential memory $\bar{m}$.

3.D Hierarchy of Attackers

Using Definition 3.1, we define a stronger notion of information-flow preservation.

Definition 3.3. Given an attacker observing n bits of memory, a program p is less secure than a program p' (written $p \preccurlyeq_n p'$) if their respective Attacker Knowledge are ordered such that $\forall c, m, m'$ we have:

$$\left.\begin{array}{c}(p,c) \Downarrow m \\ \wedge \\ (p',c) \Downarrow m'\end{array}\right\} \Rightarrow \forall o'.\exists o.\mathcal{K}_n^m(p,o) \subseteq \mathcal{K}_n^{m'}(p',o')$$

Definition 3.3 requires an alternation of quantifiers to model the fact that the Attacker of program p' chooses the partial observation o' it intends to perform and challenges the Attacker of program p to perform a more powerful observation o with a smaller knowledge set than the observation o'. Note also that if the observations are total ($n = card(Addr)$), Definition 3.3 coincides with Definition 2.2 because we necessarily have $o' = m'$ and $o = m$.

Using the previous ordering of programs, we are ready to define our notion of information-flow preservation.

Definition 3.4 (Information Flow Preserving Transformation). A program transformation $T \in Program \to Program$ is Information Flow Preserving (written ($IFP(T)$)) if and only if we have

$$\forall p \in Program, \bigwedge_{n\in\mathbb{N}} (p \preccurlyeq_n T(p)).$$

Example 3.5 explains how Definition 3.4 ensures that the program transformation of Figure 4 is deemed insecure.

Example 3.5. Figure 5 depicts the case where a 1-bit observation by an attacker on the memory of `f'` cannot be matched by a 1-bit observation on the memory of `f`.

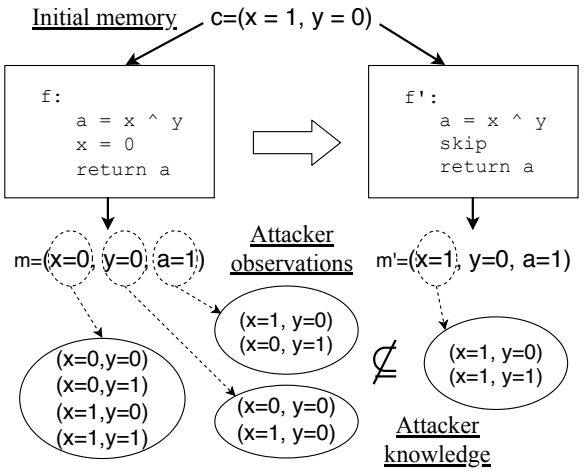

Figure 5: not *IFP* for an attacker with 1 bit of observation

As before, consider that both programs are run from the initial configuration $c = (x = 1, y = 1)$. In `f'`, the attacker performs a 1-bit observation and picks the value of `x` *i.e.*, the key that is not erased. From this observation, she gets perfect knowledge of the value of `x` but obviously no information about `y`.

$$\mathcal{K}_1^{m'}(f', (x = 1)) = \{(x = 1, y = 0); (x = 1, y = 1)\}.$$

To prove that the transformation is secure, Definition 3.4 mandates that there should exist a stronger 1-bit observation for program `f`. As shown by Figure 5, this is not possible. We examine in turn each of the possible 1-bit observations.

- As the variable `x` is overwritten, no knowledge at all is derived from its observation
$$\mathcal{K}_1^m(f, (x = 0)) = Config.$$
- Observing `y` gives no knowledge over `x`
$$\mathcal{K}_1^m(f, (y = 0)) = \{(x = 0, y = 0), (x = 1, y = 0)\}.$$
- Observing `a` gives some information about the possible initial configurations. Yet, the knowledge set is not included in $\mathcal{K}_1^{m'}(f', (x = 1))$.
$$\mathcal{K}_1^m(f, (a = 1)) = \{(x = 1, y = 0), (x = 0, y = 1)\}.$$

As a result, the transformation is not Information Flow Preserving.

4 SUFFICIENT CONDITIONS FOR IFP

The next step is to identify a set of sufficient conditions to prove that a program transformation is IFP (Section 4.A and Section 4.B) and use them to study two compiler optimisations: Dead Store Elimination (Section 5.A) and Register Allocation (Section 5.B).

4.A Equivalent Programs

As our IFP property is an end-to-end property, it is independent of syntactic differences between programs. As a result, if two programs always compute the same memories, the transformation is IFP.

Definition 4.1 (Equivalent programs). Two programs p and p' are equivalent (written $p \equiv p'$) if for every configuration $c \in Config$ they compute the same memory. Formally, we have:

$$p \equiv p' \Leftrightarrow \forall c, m, m'.((p,c) \Downarrow m) \wedge ((p',c) \Downarrow m') \Rightarrow m = m'$$

Theorem 4.2 states that a transformation mapping a program to an equivalent program is IFP.

Theorem 4.2. *Let $T : Program \to Program$. We have*

$$(\forall p, p \equiv T(p)) \Rightarrow IFP(T)$$

Proof. By Definition 3.3 and Definition 3.4, we have to prove that:

$$\left.\begin{array}{c}(p,c) \Downarrow m \\ \wedge \\ (T(p),c) \Downarrow m'\end{array}\right\} \Rightarrow \forall o'.\exists o.\mathcal{K}_n^m(p,o) \subseteq \mathcal{K}_n^{m'}(T(p),o')$$

First, notice that for two equivalent programs, the Knowledge of the attacker is the same, *i.e.*, we have

$$\mathcal{K}_n^m(p,o') = \mathcal{K}_n^m(T(p),o'). \quad (1)$$

For every observation o', we take o to be o'. It then remains to prove that

$$\mathcal{K}_n^m(p, o') \subseteq \mathcal{K}_n^{m'}(T(p), o').$$

As $p \equiv T(p)$, we have $m = m'$ and by Equation 1 the property holds. □

4.B Constant and Matching Addresses

Definition 3.3 relates program executions using the general notion of Attacker Knowledge. In practice, it is often more convenient to characterise explicitly the relation between the produced memories before and after an IFP transformation.

Note that a memory address whose value is always constant contains no knowledge about the input configuration. This remark is useful to restrict the relation between memories to only relevant (*i.e.*, non constant) addresses.

Definition 4.3 (Constant address). For a transformed program p', the set of constant addresses is the set of addresses which return the same value for every execution of the program. Formally, we have:

$$cst(p') = \{a \mid \exists v. \forall c. (p', c) \Downarrow m \Rightarrow m(a) = v\}$$

A second remark is that if two addresses a and a' always contain the same value, their knowledge is identical and therefore an observation of a' can be matched with an observation of a.

Definition 4.4 (Matching Address). Given two programs p and p', the addresses a and a' are in matching correspondence (written $p_a \equiv_{a'} p'$) if for any execution, they contain the same value. Formally, we have:

$$p_a \equiv_{a'} p' \Leftrightarrow \forall c. (p, c) \Downarrow m \wedge (p', c) \Downarrow m' \Rightarrow m'(a') = m(a)$$

By combining Definition 4.3 and Definition 4.4, we obtain a sufficient condition for an IFP transformation. Intuitively, a transformation is IFP if for every address a' that is not constant for p' there exists a matching address a in p. This is stated formally in Theorem 4.5.

Theorem 4.5. *The transformation from program p to program p' is IFP if*

$$(\forall a'. a' \notin cst(p') \Rightarrow \exists a. p_a \equiv_{a'} p') \Rightarrow \forall n. p \preccurlyeq_n p'.$$

Proof. A comprehensive proof can be found in Appendix A. The following proof sketch presents the main arguments. By expressing our theorem in terms of *attacker knowledge* our goal is to prove the following:

$$\forall n. \forall o'. \exists o. \mathcal{K}_n^m(p, o) \subseteq \mathcal{K}_n^{m'}(p', o')$$

We proceed by induction on n:

• $\mathbf{n = 0}$. This part is straightforward since the attacker can not observe any bits of the returned memory. The only observation that can be made by the attacker is $\perp$, the observation with an empty domain. Hence the attacker knowledge we obtain is the set of all the possible inputs *Config*.

$$\forall\, p\ m,\ \mathcal{K}_0^m(p, \perp) \;=\; Config$$

With this relation we can easily prove our goal for $n = 0$.

• $\mathbf{n \to n + 1}$. We slightly simplify the notation

$$\bigcup_{o \sqsubseteq m} \mathcal{K}(p, m) \to \bigcup_{o} \mathcal{K}(p, m)$$

for readability. The relation $o \sqsubseteq m$ is implicit with the memory used as the second argument of attacker knowledge.

A relation between attacker knowledge for n and $n + 1$ exists and is the following (more details in Appendix A):

$$\mathcal{K}_{n+1}^{m[a \to v]}(p, o[a \to v]) = \mathcal{K}_n^{m[a \to v]}(p, o) \setminus \bigcup_{o} \mathcal{K}(p, \bar{m}[a \nrightarrow v])$$

We define $m[a \to v]$ as a memory satisfying $m(a) = v$. The observation o has domain of size n and $a \notin dom(o)$. $o[a \to v]$ is the composition of o and the function $a \to v$ hence its domain has cardinal $n + 1$. Therefore attacker knowledge of $n + 1$ on $m[a \to v]$ with observation $o[a \to v]$ is equal to the attacker knowledge of n with observation o minus the union of all the possible configurations leading to a potential memory $\bar{m}$ satisfying both $o \sqsubseteq \bar{m}$ and $\bar{m}(a) \neq v$.

Starting from our goal proof we develop the following proposition with the help of our induction hypothesis:

$$\begin{array}{lcl}
\mathcal{K}_{n+1}^{m[a \to v]}(p, o[a \to v]) & \subseteq & \mathcal{K}_{n+1}^{m'[a' \to v']}(p', o'[a' \to v']) \\
\Leftrightarrow & & \\
\left(\begin{array}{c} \mathcal{K}_n^{m[a \to v]}(p, o) \\ \setminus \\ \bigcup_{o} \mathcal{K}(p, \bar{m}[a \nrightarrow v]) \end{array}\right) & \subseteq & \left(\begin{array}{c} \mathcal{K}_n^{m'[a' \to v']}(p', o') \\ \setminus \\ \bigcup_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v']) \end{array}\right) \\
\Leftrightarrow & & \\
\bigcup_{o} \mathcal{K}(p, \bar{m}[a \nrightarrow v]) & \supseteq & \bigcup_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v'])
\end{array}$$

Recall that in our theorem we have the hypothesis that a' is either a constant or a matching address. We prove the proposition in both cases:

• $\mathbf{a' \in cst(p')}$. In this case we know that v' is a constant such that for any input the returned value at address a' is always v'. This implies that:

$$\bigcup_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v']) = \emptyset$$

Indeed, since for all executions the value at a' is always v', the memories referred by $\bar{m'}[a' \nrightarrow v']$ do not exist. With this equality we can easily prove our proposition.

• $\mathbf{p_a \equiv_{a'} p'}$. We know that a' is a matching address to a, therefore for every configuration of p and p' leading to m and m' we have $m(a') = m(a) = v$. We use the definition of attacker knowledge on our proposition which gives:

$$\{c \mid (p, c) \Downarrow \bar{m}[a \nrightarrow v]\} \;\supseteq\; \{c \mid (p', c) \Downarrow \bar{m'}[a' \nrightarrow v]\} \qquad (2)$$

We need to prove that for every configuration c where $(p', c) \Downarrow \bar{m'}$ with $\bar{m'}(a') \neq v$ then we also have $(p, c) \Downarrow \bar{m}$ with $\bar{m}(a) \neq v$. This is straightforward since we know that $\bar{m}(a) = \bar{m'}(a')$ then having $\bar{m'}(a') \neq v$ necessarily implies $\bar{m}(a) \neq v$. We have proved our proposition and Theorem 4.5.

□

Theorem 4.5 provide a sound characterisation of an IFP transformation. However, it is not complete. This is illustrated by the following IFP transformation $x = y \rightarrow x =\sim y$ which does not meet the pre-conditions of Theorem 4.5. Yet, the pre-conditions are sufficiently general to cover sophisticated program transformations such as Register Allocation (see Section 5.B).

5 IFP TRANSFORMATIONS

In this section we focus on two compilation steps which are not *a priori* IFP: *Dead Store Elimination* (DSE) and *Register Allocation* (RA). We show how to make these two transformations IFP and we prove them sound using two different proof techniques. First we present a modified DSE and prove directly that its algorithm is sound. Second, we show that RA can be made IFP by resorting to an *a posteriori* approach using a certified validator which checks the IFP condition for a particular program and its RA transformation.

5.A Dead Store Elimination

DSE is the archetypical program transformation that is not IFP. Informally, a dead store is a memory write which is provably unnecessary to compute the program result. Hence, from an optimisation point of view, it is a perfectly legal transformation to remove a dead store instruction. Security-wise, as shown by our motivating example (see Figure 1) and Program g of Figure 2, the transformation does not preserve information-flows. Indeed, an attacker may gain more knowledge by observing the value that is not overwritten due to the removal of a dead store.

A drastic solution would be to disable this optimisation. We propose a modified Dead Store Elimination optimisation based on a revised and strengthened notion of dead store.

5.A.1 Liveness based DSE. Dead stores are typically identified using a *liveness analysis* [15, 2.1.4]. A liveness analysis is a classic backward program analysis. For each program point, it computes an over-approximation of the *live* variables *i.e.* variables that are necessary to compute the program result. Dually, *dead* variables are those variables that are not live and a dead store is a memory write $(x = e)$ where the variable x is dead. Therefore, a classic DSE performs a liveness analysis and removes dead stores.

Our abstract model of programs does not feature program points. Instead we use the end-to-end property of a liveness analysis:

Definition 5.1 (Sound Liveness). Given a program p, an end-to-end liveness analysis is a pair of sets of addresses $(V, W) \subseteq Addr \times Addr$ such that

$$\forall c, c', m, m'. \quad \left.\begin{array}{c} (p, c) \Downarrow m \\ \wedge \\ (p, c') \Downarrow m' \end{array}\right\} \Rightarrow c \simeq_V c' \Rightarrow m \simeq_W m'$$

where

$$e \simeq_V e' \Leftrightarrow \forall x \in V. e(x) = e'(x)$$

for $(e, e') \in Config \times Config$ or $(e, e') \in Memory \times Memory$.

Definition 5.1 says that if two input configurations c and c' agree on the values of the input live addresses V then the output memories agree on the output live addresses W. For the purpose of optimisation, the set W contains a minimum set of addresses *i.e.*, only those needed by the return statement of the program; the set V being computed by a backward fixpoint iteration (see [15, 2.1.4] for details).

As DSE removes the dead statements, it keeps unchanged the values of live addresses. As a result, in our formal model, a DSE transformation can be characterised by Definition 5.2.

Definition 5.2 (Dead Store Elimination). Given a program p and a liveness analysis (V, W) of p. The DSE transformation from p to p' is correct if p' is indistinguishable from p for all the live variables W. Formally, we have:

$$\forall c. (p, c) \Downarrow m \wedge (p', c) \Downarrow m' \Rightarrow m \simeq_W m'$$

This specification is partial. In particular, it says nothing about the dead variables but this is enough to conclude that these dead variables may leak information and violate our IFP property.

5.A.2 Shadow Store Elimination. In order to get an IFP DSE, we propose to exploit Theorem 4.2 and therefore have a dead store elimination producing an equivalent program according to Definition 4.1. Interestingly, this result can be obtained in a non-intrusive way by only slightly modifying the initial condition of the liveness analysis. Indeed, it is sufficient to impose that, at the end of the program, every address is live. As liveness analysis is backward, this can always be done.

Theorem 5.3. *Given a program p and a liveness analysis (V, W) of p such that $W = Addr$, DSE produces a program p' that is IFP.*

Proof. By Theorem 4.2, it suffices to prove $p \equiv p'$. Suppose that $(p, c) \Downarrow m$ and $(p', c) \Downarrow m'$. By Definition 5.2 of a DSE transformation, we have $m \simeq_{Addr} m'$ *i.e.*, $m = m'$. As a result, Theorem 5.3 holds. □

The effect of this modification is that only dead stores that are shadowed by a following store at the same address can be safely removed. Said otherwise, for a given address, the last store needs to be retained (whether it is dead or not). However, the penultimate dead store can be safely removed as its value is overwritten later on.

5.B Register allocation

Register Allocation (RA) is an essential compiler pass which makes sure that the target program abides to the constraints of the underlying architecture. In particular, RA compiles source programs using an unlimited number of variables into target programs using a limited number of hardware registers. RA is based on sophisticated algorithms and heuristics *e.g.*, Iterated Register Coalescing [8], which aim at optimising the usage of registers. As a last resort, when no more registers are available, the RA algorithm performs a so-called *register spill i.e.*, the value of the register is temporarily saved in memory so that the register can be reused for other purpose. The symmetric operation is *register reloading* consisting in restoring the saved value.

Example 5.4. Figure 6 illustrates the working of a RA algorithm for an architecture with only two registers. The source code on the left computes the bitwise xor of `text` with some salt and some key. Note that the source code is implicitly erasing the key by reusing the variable `c` for computing the program result.

```
1 cipher(text)
2     c = get_key()
3     salt = get_salt()
4     temp = text ^ salt
5     c = c ^ temp
6     return c
```

```
1 cipher'(text)
2     r2 = get_key()
3     r1 = get_salt()
4     c = r2              //spill r2 in c
5     r2 = text           //load text in r2
6     r2 = r2 ^ r1        //compute temp
7     r1 = c              //reload c in r1
8     r1 = r1 ^ r2
9     return r1
```

Figure 6: Original program (left) After register allocation (right)

The target code on the right is the result of a typical RA algorithm. The architecture has only two registers *i.e.*, r1 and r2. Moreover, the architecture imposes that a xor operation can only be performed with register operands.

Initially, the allocator maps the variable c to the register r2 and the variable salt to the register r1. However, to perform a xor with text, the allocator needs to reuse a register. To do that, the register r2 is spilled in memory in the variable c so that r2 can be reused for storing the program argument text. For the second xor, the variable c is reloaded in register r1.

From a security point of view, the register allocation of Figure 6 is not an IFP transformation. The culprit is the spilling of register r2. Whereas the value of r2 is eventually overwritten, its copy remains in memory in variable c. Compared to DSE, the introduction of the information flow leak is not due to an aggressive optimisation but is really needed for a successful RA.

To ensure and enforce that RA is an IFP transformation we propose a variation of the translation validation approach [16].

5.B.1 Translation Validation of Register Allocation. The design of our validator is guided by Theorem 4.5 which states sufficient conditions for an IFP transformation. Definition 5.5 states the soundness conditions for such a validator.

Definition 5.5 (Sound Validator). Let p be a program and p' be a transformed program. A sound validator is such that $IFPok(p, p') = true$ entails that

$$\forall a'.a' \notin cst(p') \Rightarrow \exists a.p_a \equiv_{a'} p'$$

In other words if a' is not a constant address of p', then there must exist a matching address a in p. As the conditions quantify over all addresses, a naive approach *i.e.*, enumerating over every address, is not feasible. To get an effective solution, notice that RA keeps the memory mostly unchanged with the exception of a limited number of locations which are program variables, registers and spilled locations. For our formal model, consider that the set of addresses is partitioned into a set of named addresses *NAddr*, affected by register allocation, and unamed addresses *UAddr* untouched by register allocation. As a result, the validator needs to establish a precise mapping between named addresses but only needs to check that other addresses have the exact same value before and after the transformation.

The validator performs a constant analysis of the transformed program thus identifying addresses which do not leak any information. Constant analysis is a standard program analysis. For our context, Definition 5.6 states the soundness guarantee of such a constant analysis.

Definition 5.6 (Soundness of Constant Analysis). For a program p, a sound constant analysis $cst^\sharp(p) \in NAddr \rightharpoonup \mathbb{B}$ is a partial function mapping named addresses to a constant values such that

$$dom(cst^\sharp(p)) \subseteq cst(p)$$

A matching analysis is less standard but existing validators for RA [9, 17] implement a similar technique. Definition 5.7 states the soundness requirements.

Definition 5.7 (Sound Matching Analysis). For programs p and p', a sound matching analysis

$$match^\sharp \in Program \times Program \rightharpoonup (NAddr \rightharpoonup NAddr)$$

is such that if $match^\sharp(p, p')$ is defined we have that

- all the unnamed addresses have the same value in both p and p' ($\forall a' \in UAddr, p_{a'} \equiv_{a'} p'$);
- and the obtained mapping $\varphi \in NAddr \rightharpoonup NAddr$ relates matching addresses ($\forall a' \in NAddr, p_{\varphi(a')} \equiv_{a'} p'$).

Using the previous analyses, we can construct a sound IFP validator for RA.

Theorem 5.8 (Sound Validator). *Let p be a program and p' obtained by an untrusted RA. Suppose a RA validator defined by*

$$\begin{aligned} IFPok(p, p') \quad = \quad & \exists \varphi. match^\sharp(p, p') = \varphi \quad \wedge \\ & \forall a' \notin cst^\sharp(p').\exists a.\varphi(a') = a \end{aligned}$$

If $IFPok(p, p') = true$ then the transformation is IFP.

Proof. By Theorem 4.5, it amounts to proving

$$\forall a'.a' \notin cst(p') \Rightarrow \exists a.p_a \equiv_{a'} p'$$

By hypothesis, the matching analysis succeeds such that $match^\sharp(p, p') = \varphi$ for some φ. The proof is case analysis over the address a'.

- If $a' \in UAddr$, by Definition 5.7 and because the validator succeeds, we have $p_{a'} \equiv_{a'} p'$ and therefore the property holds.
- If $a' \in NAddr \wedge a' \in dom(cst^\sharp(p'))$. By Definition 5.6, we have $a' \in cst(p')$ and, by contradiction, the property holds.
- If $a' \in NAddr \wedge a' \notin cst^\sharp(p')$, by Definition 5.7, we have $p_{\varphi(a')} \equiv_{a'} p'$ and the property holds.

□

Example 5.9 illustrates the running of a matching analysis.

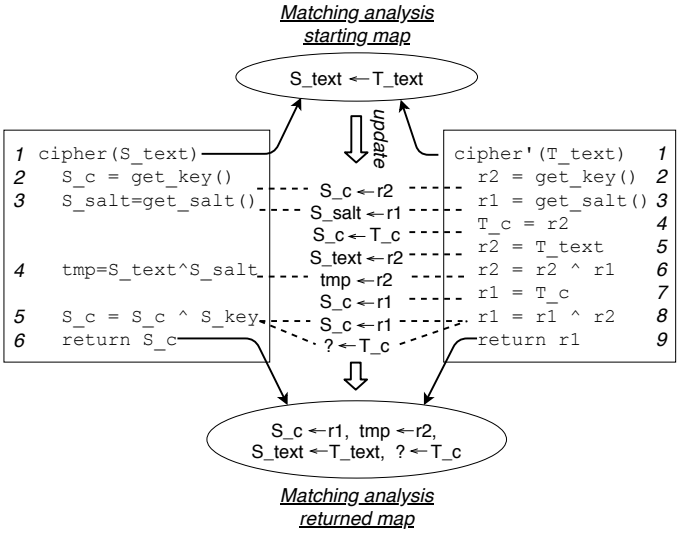

Figure 7: Matching analysis example

Example 5.9. The matching analysis for the programs of Figure 6 is shown Figure 7. For clarity, variable names are prefixed by a S_ in the source code and by a T_ in the target code. A matching is attached to a pair of program points and we write $\varphi_{i,j}$ for a matching φ where i is a program point of the source program and j is a program point of the target program. As both programs execute, the matching is updated. In Figure 7, a dashed line links an update of the mapping with the instruction responsible for the update.

At the start of both programs the parameters are stored at the same address. As the result, we have the initial matching $\varphi_{1,1} = [T\_text \mapsto S\_text]$. After the first two instructions, we have the matching $\varphi_{3,3} = [r_2 \mapsto S\_C, r_1 \mapsto S\_salt, T\_text \mapsto S\_text]$. The next two instructions of the target code correspond to the spilling of register r_2 and loading of variable T_text and have no counterpart in the source code. Therefore, we get the matching $\varphi_{3,5} = [r_2 \mapsto S\_text, r_1 \mapsto S\_salt, T\_c \mapsto S\_c, T\_text \mapsto S\_text]$. At the entry of the first xor, the operator and the operands do match and after the instruction the matching is updated and we get:

$$\varphi_{4,6} = [r_2 \mapsto tmp, r_1 \mapsto S\_salt, T\_c \mapsto S\_c, T\_text \mapsto S\_text]$$

At the instruction 8 of the transformed program, the register r_1 is mapped to S_C. However, as the source variable S_c is overwritten, the variable T_c has no more mapping with a source variable hence we remove it from φ. We get the final mapping

$$\varphi_{5,9} = [r_2 \mapsto tmp, r_1 \mapsto S\_c, T\_text \mapsto S\_text]$$

At the end of program, the variable T_c has no mapping. This is the witness of the information flow leak and the reason why the validator rejects the transformation.

5.B.2 From Translation Validation to IFP RA. As register allocation is, in general, not IFP, there is very little chance that our validator would ever succeed. However, as illustrated by Example 5.9, the missing mappings of the validator indicate the information flow leaks. Closing them can be done by zeroing all those variables or registers at the end of the program. For the program of Figure 7, this amounts to adding the assignment T_c = 0. After the transformation, the matching analysis would be unchanged. However, the constant analysis would detect that the value of $T\_c \mapsto 0$ *i.e.*, leaks no information, and therefore prove that the transformation is IFP.

This approach is non-intrusive and compatible with any RA algorithm. Patching the information leaks *a posteriori* may not be the most efficient approach and an IFP aware RA algorithm could make a wiser reuse of spilling locations. Yet, practical experiments are needed to assess the effect on the quality of the generated code. In the worst case, all spilled locations and registers would need to be overwritten. In practice, as calling conventions mandate callee-saved registers to be restored this worst-case is fortunately never reached.

6 DISCUSSION

For the sake of clarity, our formal model is simple. Yet, the result of this paper could be generalised to more complex contexts *e.g.*, where programs executions are not necessarily terminating and the attacker is not limited to a single observation at the end of the program.

For Termination, Definition 3.3 takes as hypothesis that both programs terminate and is therefore *termination sensitive*. If a non-terminating program p is transformed into a terminating program p', the property is vacuously true whereas the transformed program potentially leaks more information. To deal with this issue, it is sufficient to require that p' only diverges if p already does. As most compiler optimisations preserve termination, this is a mild requirement which always holds in practice.

For the Attacker model, our assessment is that an omniscient Attacker able to observe memory at any time would preclude too many optimisations. A more practical scenario would be to define pre-determined observation points, *e.g.*, function exits, which can be matched between a program and its transformation. In this case, our IFP definition would need to be generalised. In particular, if the attacker possesses a memory, each observation would monotonically increase its knowledge.

Even after refinement, our attacker model is still quite strong and does not model how difficult it is to precisely probe the memory. For instance, it considers equally insecure a program leaking a key spread over the whole memory or the same key stored in adjacent memory locations. Similarly, for our attacker model, duplicating (secret) information does not increase the information flow leak whereas, intuitively, it makes practical attacks more feasible. An interesting extension would be to equip the attacker with probability distributions modelling how hard it is to successfully and precisely read a given memory location.

7 RELATED WORK

The problem of secure erasure of secrets has been studied from an information-flow perspective. Chong and Myers [4] propose semantics foundations for defining erasure policies. A main insight behind that work is that erasure can be seen as the dual of declassification [18]. Later on, Chong and Myers [5], but also Hunt and Sands [10], propose type-systems for verifying erasure policies of

the source code. Askarov *et al.* enforce the erasure policies in the presence of so-called write-once locations which cannot be overwritten. The key insight is to store encrypted data in write-once locations and simulate erasure by the erasure of the cryptographic key. In our work, we do not consider erasure policies but ensure that program transformations do not introduce information leaks. Our model of attacker is currently too idealised to ensure that IFP transformations preserve the rich erasure policies defined by the aforementioned works. An attacker observing all the intermediate memory states would probably suffice but would preclude too many transformations.

Recently, D'Silva *et al.* [7] drew the attention to the *correctness-security* gap, explicitly mentioning dead-store elimination as a problematic transformation. Deng and Namjoshi [6] ensure that a modified DSE transformation preserves their information flow notion of *leaky triple*. This notion does not capture the amount of information leak. For instance, it ensures that a non-interferent program is transformed into another non-interferent program but allows a leaky program to be transformed into a more leaky program. Our IFP property, based on the notion of Attacker Knowledge, is stronger and rules out this possibility: it ensures that the transformed program is no more leaky. Yang *et al.* [23] survey the workarounds used by developers to prevent compilers to remove security-sensitive dead-stores. They also propose a secure DSE implementation for LLVM which verifies our IFP property. Simon, Chisnall and Anderson [19] investigate how compilers may break the security of cryptographic code. They propose compiler support for the secure erasure of secret which improves the security of the generated code.

Recently, Barthe, Grégoire and Laporte [2] have proposed a general framework to reason about the preservation of information leakage. They use an instrumented semantics which explicitly leaks information to the attacker and propose reasoning principles based on so-called 2-simulations. It is unclear whether our attacker model fits their framework because our attacker performs an arbitrary, non-deterministic, observation of the memory. Our reasoning principles are simpler because our necessary conditions for IFP (see Theorem 4.5) can be enforced using traditional simulation proof techniques for compiler correctness [14].

8 CONCLUSION

Compiler optimisations introduce information leaks which make them unreliable for security crtitical code. In this work, we propose a formal definition of Information Flow Preserving (IFP) transformation using an attacker with the ability to observe memory. A transformation is IFP if an attacker does not gain any advantage by observing the final memory after running the transformed program. We have shown that classic optimisations such as Dead Store Elimination (DSE) and Register Allocation (RA) are in general not IFP transformation. To our knowledge, this is the first time that DSE is shown insecure based on an Information Flow argument. The good news is that, for DSE and RA, modifying the transformations to be IFP is feasible in a rather non-intrusive way.

In theory, optimisations that improve the instruction sequence but preserve memory writes (*e.g.* constant propagation or common expression elimination) are IFP transformations. As future work, we intend to investigate more optimisations and review their actual implementation. On a more practical side, we intend to evaluate the difficulty of modifying a realistic compiler to make it IFP but also the loss of efficiency this modification may incur. Another limitation to address is that we only consider whole-program optimisations. Further work should examine to what extent it is possible to obtain a more compositional model that can reason about individual functions and in which an attacker can observe memory in-between function calls.

ACKNOWLEDGMENTS

This work is partially funded by the grant ANR-14-CE28-0014.

REFERENCES

[1] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies *(SP '07)*. IEEE Computer Society, 207–221. https://doi.org/10.1109/SP.2007.22

[2] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *Proc. of the 31st Computer Security Foundations Symposium*. IEEE Computer Society.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

[4] Stephen Chong and Andrew C. Myers. 2005. Language-Based Information Erasure. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFW '05)*. IEEE Computer Society, Washington, DC, USA, 241–254. https://doi.org/10.1109/CSFW.2005.19

[5] Stephen Chong and Andrew C. Myers. 2008. End-to-End Enforcement of Erasure and Declassification. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium (CSF '08)*. IEEE Computer Society, Washington, DC, USA, 98–111. https://doi.org/10.1109/CSF.2008.12

[6] Chaoqiang Deng and Kedar S. Namjoshi. 2016. Securing a Compiler Transformation. In *23rd Int. Static Analysis Symposium (LNCS)*, Vol. 9837. Springer, 170–188.

[7] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proc. of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, 73–87. https://doi.org/10.1109/SPW.2015.33

[8] Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (1996), 300–324. https://doi.org/10.1145/229542.229546

[9] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. 2006. Catching and Identifying Bugs in Register Allocation. In *Proceedings of the 13th International Conference on Static Analysis (SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 281–300. https://doi.org/10.1007/11823230_19

[10] Sebastian Hunt and David Sands. 2008. Just Forget It: The Semantics and Enforcement of Information Erasure. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems (ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 239–253. http://dl.acm.org/citation.cfm?id=1792878.1792903

[11] ISO. 20111. *ISO C Standard 2011*. Technical Report.

[12] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. of the 3d Int. Symp. on NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 41–55.

[13] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 42–54. https://doi.org/10.1145/1111037.1111042

[14] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[15] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.

[16] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems (LNCS)*, Vol. 1384. Springer, 151–166. https://doi.org/10.1007/BFb0054170

[17] Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 224–243. https://doi.org/10.1007/978-3-642-11970-5_13

[18] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548. https://doi.org/10.3233/JCS-2009-0352

[19] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *3rd IEEE European Symposium on Security and Privacy (EuroSP)*. IEEE.

[20] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/2951913.2951924

[21] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems (APSys '12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2387841.2387850

[22] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[23] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 1025–1040. http://dl.acm.org/citation.cfm?id=3241189.3241269

A PROOF OF THEOREM 4.5

We imagine an execution of p and p' resulting in memories m and m' respectively. We keep for later use the hypothesis on p and p':

$$\forall a'.a' \notin cst(p') \Rightarrow \exists a.p_a \equiv_{a'} p' \tag{H}$$

Our goal is to prove:

$$\begin{array}{lll} & \forall n. & p \preccurlyeq_n p' \\ \Leftrightarrow & \forall n. \forall o. \exists o'. & \mathcal{K}_n^m(p,o) \subseteq \mathcal{K}_n^{m'}(p',o') \end{array}$$

We do an induction on $n \in \mathbb{N}$.

• $\mathbf{n = 0}$. This part is straightforward since the attacker can not observe any bits of the returned memory. The only observation that can be made by the attacker is $\perp$, the observation with an empty domain. Hence the attacker knowledge we obtain is the set of all the possible inputs *Config*.

$$\forall\ p\ m,\ \mathcal{K}_0^m(p,\perp)\ =\ Config$$

With this relation we can easily prove our goal for $n = 0$.

• $\mathbf{n \to n+1}$. The remaining part to prove for our induction is that if our theorem is true for n it is also true for $n+1$. Before describing the hypothesis and goals we search for a relation between attacker knowledge observing n and $n+1$ bits.

We use the notation $o[a \to v]$ for an observation of $n+1$ bits composed of the observation of n bits o and the observation of one bit at address a which returns v.

We detail the definition of an attacker knowledge of n bits on the memory $m[a \to v]$ where $m(a) = v$. The attacker knowledge is built with the observation o where we assume $a \notin dom(o)$. We slightly simplify the notation from $\bigcup_{o \sqsubseteq m} \mathcal{K}(p,m)$ to $\bigcup_o \mathcal{K}(p,m)$ for readability. The relation $o \sqsubseteq m$ is implicit with the memory used as the second argument of attacker knowledge.

$$\mathcal{K}_n^{m[a\to v]}(p,o) \quad = \quad \bigcup_o \mathcal{K}(p,\bar{m}) \tag{1}$$

$\Leftrightarrow$

$$\mathcal{K}_n^{m[a\to v]}(p,o) \quad = \quad \left(\begin{array}{c} \bigcup_o \mathcal{K}(p,\bar{m}[a \to v]) \\ \cup \\ \bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v]) \end{array} \right) \tag{2}$$

$\Leftrightarrow$

$$\left(\begin{array}{c} \mathcal{K}_n^{m[a\to v]}(p,o) \\ \setminus \\ \bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v]) \end{array} \right) \quad = \quad \bigcup_o \mathcal{K}(p,\bar{m}[a \to v]) \tag{3}$$

$\Leftrightarrow$

$$\left(\begin{array}{c} \mathcal{K}_n^{m[a\to v]}(p,o) \\ \setminus \\ \bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v]) \end{array} \right) \quad = \quad \bigcup_{o[a \nrightarrow v]} \mathcal{K}(p,\bar{m}) \tag{4}$$

$\Leftrightarrow$

$$\left(\begin{array}{c} \mathcal{K}_n^{m[a\to v]}(p,o) \\ \setminus \\ \bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v]) \end{array} \right) \quad = \quad \mathcal{K}_{n+1}^{m[a\to v]}(p,o[a \to v]) \tag{5}$$

Equation (1) is simply the definition of Attacker Knowledge. $\bar{m}$ is the possible memories which can have o as a partial function: $o \sqsubseteq \bar{m}$.

From (1) to (2) we split Attacker Knowledge into two. First the configurations whose execution results in memory $\bar{m}$ where the value at a is v ($\bar{m}[a \to v]$). Second the memories where the value at a is different from v ($\bar{m}[a \nrightarrow v]$).

To go from (2) to (3) we pass the term $\bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v])$ to the left side of the equality turning the operator $\cup$ into $\setminus$. This is true because $\bigcup_o \mathcal{K}(p,\bar{m}[a \nrightarrow v])$ and $\bigcup_o \mathcal{K}(p,\bar{m}[a \to v])$ are disjoint sets. Indeed since our programs are deterministic the execution of a configuration c can not lead to both $\bar{m}[a \to v]$ and $\bar{m}[a \nrightarrow v]$.

From (3) to (4) we use the equality:

$$\bigcup_o \mathcal{K}(p,\bar{m}[a \to v]) = \bigcup_o \mathcal{K}(p,\bar{m})$$

Instead of considering memories $\bar{m}[a \to v]$ that satisfy both $o \sqsubseteq \bar{m}$ and $\bar{m}(a) = v$, we only consider memories that satisfy $o[a \to v] \sqsubseteq \bar{m}$. For reminder, $o[a \to v]$ is the composition of o and the function that associates a to the value v. Therefore the equality is true since memories satisfying $o[a \to v] \sqsubseteq \bar{m}$ also satisfy the two conditions on $\bar{m}[a \to v]$.

From (4) to (5) we only apply the definition of attacker knowledge for an observation $o[a \to v]$ of $n+1$ bits.

With this new relation we can prove that if the property is true for n then it is also true for $n+1$. We develop the following proposition with the help of the induction hypothesis (H):

$$\begin{array}{lcl} \mathcal{K}_{n+1}^{m[a\to v]}(p, o[a \to v]) & \subseteq & \mathcal{K}_{n+1}^{m'[a'\to v']}(p', o'[a' \to v']) \\ \Leftrightarrow & & \\ \left(\begin{array}{c} \mathcal{K}_{n}^{m[a\to v]}(p, o) \\ \setminus \\ \bigcup\limits_{o} \mathcal{K}(p, \bar{m}[a \nrightarrow v]) \end{array}\right) & \subseteq & \left(\begin{array}{c} \mathcal{K}_{n}^{m'[a'\to v']}(p', o') \\ \setminus \\ \bigcup\limits_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v']) \end{array}\right) \\ \Leftrightarrow & & \\ \bigcup\limits_{o} \mathcal{K}(p, \bar{m}[a \nrightarrow v]) & \supseteq & \bigcup\limits_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v']) \end{array}$$

We remind that in our theorem we have the hypothesis that a' is either a constant or a matching address. We prove the proposition in both cases:

• $\mathbf{a'} \in \mathbf{cst(p')}$. In this case we know that v' is a constant such that for any input the returned value at address a' is always v'. Meaning that:

$$\bigcup_{o'} \mathcal{K}(p', \bar{m'}[a' \nrightarrow v']) = \emptyset$$

Indeed since for all executions the value at a' is always v', the memories referred by $\bar{m'}[a' \nrightarrow v']$ do not exist. With this equality we can easily prove our proposition.

• $\mathbf{p_a} \equiv_{\mathbf{a'}} \mathbf{p'}$. We know that a' is a matching address to a therefore for every configuration of p and p' leading to m and m' we have $m(a') = m(a) = v$. We use the definition of attacker knowledge on our proposition which gives:

$$\{c \mid (p, c) \Downarrow \bar{m}[a \nrightarrow v]\} \supseteq \{c \mid (p', c) \Downarrow \bar{m'}[a' \nrightarrow v]\} \qquad (3)$$

We need to prove that for every configuration c where $(p', c) \Downarrow \bar{m'}$ with $\bar{m'}(a') \neq v$ then we also have $(p, c) \Downarrow \bar{m}$ with $\bar{m}(a) \neq v$. This is straightforward since we know that $\bar{m}(a) = \bar{m'}(a')$ then having $\bar{m'}(a') \neq v$ necessarily implies $\bar{m}(a) \neq v$. We have proved our goal and Theorem 4.5.