2019 IEEE 32nd Computer Security Foundations Symposium (CSF)

Information-Flow Preservation in Compiler Optimisations

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

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

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

***Abstract*—Correct compilers perform program transformations preserving input/output behaviours of programs. However, correctness does not prevent program optimisations from introducing information-flow leaks that would make the target program more vulnerable to side-channel attacks than the source program. To tackle this problem, we propose a notion of Information-Flow Preserving (IFP) program transformation which ensures that a target program is no more vulnerable to passive side-channel attacks than a source program. To protect against a wide range of attacks, we model an attacker who is granted arbitrary memory accesses for a pre-defined set of observation points. We propose a compositional proof principle for proving that a transformation is IFP. Using this principle, we show how a translation validation technique can be used to automatically verify and even close information-flow leaks introduced by standard compiler passes such as dead-store elimination and register allocation. The technique has been experimentally validated on the COMPCERT C compiler.**

***Index Terms*—Secure compilation; Side-Channel; Information-Flow**

I. INTRODUCTION

When it comes to security, compilers can not be trusted. They may perform aggressive optimisations which may exploit undefined behaviours [30] or they may have subtle bugs and generate invalid code [31]. Compilers may turn secure constant-time source programs [2], [8] into target programs vulnerable to timing attacks [17]. Compilers may also increase the lifetime of sensitive data in memory and therefore make the target program more vulnerable to attackers with physical access. The pitfalls of so-called *safe erasure* in the presence of compiler optimisations is recognised by the CERT at CMU [1] and protections have recently been proposed *e.g.,* [28], [32]. The overall consequence of this unfortunate state-of-affairs is that for highly-critical code, manual inspection of the generated code is standard industrial practice. Furthermore, even if the proposed counter-measures are effective in practice, they do not come with a formal security guarantee.

Much progress has been made in the formal verification of compiler correctness, providing a semantic guarantee that the behaviour of the compiled program is compliant with the behaviour of the source code. Prominent examples are the COMPCERT C compiler [21], verified using the Coq proof assistant [23], CAKEML [19], an ML compiler verified using HOL4 [1] or seL4 [27] a verified micro-kernel whose compilation was proven correct through translation validation [24]. However, compiler correctness does not guarantee the preservation of certain non-functional properties which are essential for the security of the compiled code [14].

In this paper, we propose an end-to-end security property to ensure that a target program does not leak more information than a source language. We build on our previous work [7] but enrich the model to a more realistic setting where an attacker may observe intermediate program states and a strengthened security property preventing the *duplication* of potentially sensitive data. We consider a threat model in which an attacker is granted physical memory access at specific observation points, and is parametrised by the amount of information he is allowed to read. In our setting, a program transformation is secure if an attacker at the target language does not have an advantage with respect to an attacker at the source language. This is formalised using the standard information-flow notion of Attacker Knowledge [5].

The technical contributions of the paper can be summarised as follows:

- A formal definition of Information-Flow Preserving (IFP) transformation based on the notion of Partial Attacker knowledge.
- A simulation-based proof principle for proving that a program transformation is IFP.
- A translation validation algorithm which can be instrumented to automatically close information-flow leaks in case the transformation is not IFP.
- An implementation and experiments with an IFP register allocation pass within the COMPCERT compiler.

The rest of the paper is organised as follows. Section II explains the rationale for our attacker model. Examples illustrating how compiler optimisations may weaken security are presented in Section III. Section IV defines formally our notion of Information-Flow Preservation. In Section V, we introduce a necessary condition for IFP; this condition takes the form of a simulation-based principle. We use this principle in Section VI to design a translation validation approach for the Register Allocation pass of COMPCERT and present our experimental results in Section VII. We compare with related work in Section IX and Section X concludes.

[1]CERT MSC06-C

DOI 10.1109/CSF.2019.00023

II. Attacker Model

Our overall goal is to prevent information leaks from being introduced by the compilation process. Compiled code should be no more vulnerable to passive side-channel attacks than the source code. Such side channels correspond to an attacker who is granted physical memory access at specific observation points, and who is parametrised by the amount of information he is allowed to read. At the semantic level, side-channels can be modelled using a leakage function exposing a partial view of the program state to the attacker [6].

A typical example of a timing channel is the leaking of information through observations of the memory cache behaviour. Formally, this channel can be modelled by leaking the program counter and the memory accesses [2]. Another example is the power channel that can be modelled by the so-called Hamming Weight model [18] where what is leaked is the Hamming Weight of the program state *i.e.*, the number of information bits that are set to 1.

In our model, in order to protect against an arbitrary side-channel, we quantify over a hierarchy of attackers A_n who have access to the programs code and where n is the number of bits of information that the attacker is allowed to extract from the program memory during the program execution. Thus, we do not fix one leakage function *a priori* but rather consider protecting against a hierarchy of leakage functions.

This attacker model is sufficient to formally capture the leakage of information. However, it does not take into account the practical difficulty of a physical attack *i.e.* how hard it is to extract bits of information when the memory is indirectly observed through a noisy side-channel. For instance, an attacker would in practice have an advantage if sensitive data is duplicated multiple times across the memory. We address this problem by forbidding duplication of data. This, in turn, leads to the formal requirement that for a given information leakage, there is an injection between the target attacker observations and the source attacker observations.

Our attacker model and our notion of Information-Flow Preserving transformation are formally presented in Section IV. First, we present a few typical program transformations and explain informally whether they are Information-Flow Preserving or not.

III. Information Flow Preservation by Examples

In Figure 1, we present a list of simple programs using the syntax of an imperative language where a $\bullet$ indicates program points where the program memory is leaked to the attacker. In terms of input/output semantics, the program are indistinguishable: they all return 0. As a result, they could all be compiled from the original canonical program f that sets the variable x to 0 and returns 0. The program f leaks its initial memory ($\bullet_1$) and final memory ($\bullet_2$) to the attacker. In Figure 1, we mark the programs that are IFP with a ✓ and those which are not with a ✗. We also write $v\bullet_i$ for the observation of the value of the variable v at the leakage point $\bullet_i$. Consider Program g obtained by removing the assignment to the variable x from Program f. From a compiler point

Original program:

f: $\bullet_1$ x=0; $\bullet_2$ **return** 0;

Dead store elimination:

g: $\bullet_1$ skip; $\bullet_2$ **return** 0; ✗

Overwrite with constant value:

h: $\bullet_1$ x=1; $\bullet_2$ **return** 0; ✓

Overwrite with leaked value:

i: $\bullet_1$ x=y; $\bullet_2$ **return** 0; ✗

Leaking through direct flow:

j: $\bullet_1$ a=x; x=0; $\bullet_2$ **return** 0; ✗

Leaking through indirect flow:

k: $\bullet_1$ {**if**(x) a=0 **else** a=1}; x=0; $\bullet_2$ **return** 0; ✗

Fig. 1. Example of compiled programs

of view, this is a correct transformation because the value of x does not contribute to the result. However, removing such a *dead store* introduces an information-flow leak and should therefore be ruled out. Intuitively, the principle for a transformation to be safe is that any observation of a value made with the target program must have a corresponding observation in the source code. In particular, the observation $x\bullet_2$ in g needs to be matched by an observation of $x\bullet_2$ in f. As the observation of $x\bullet_2$ in g has no counterpart in f for any $v\bullet_2$, the transformation is rejected. Note that it is of utmost importance for the source and target observations to be synchronised *i.e.* a target observation at $\bullet_i$ needs to be matched by a source observation at the same observation point $\bullet_i$. Otherwise, Program g would be wrongly deemed secure under the fallacious argument that the observation of $x\bullet_2$ in g leaks the same information as $x\bullet_1$ in f. The synchrony of observations is our solution to make it impossible to remove or delay the erasure of some information.

Program h overwrites x with a different constant. As constants carry no information, the observation $x\bullet_2$ in f and h leak the same (empty) amount of information and therefore the transformation is information-preserving.

Program i overwrites x with the variable y. Because $y\bullet_2$ is already leaked, this does not introduce any additional information-flow leak. However, it does lead to duplication of information which we may want to forbid. In the example, the duplication may make it easier to get to the information of $y\bullet_2$ in i (it is also present in $x\bullet_2$) than in f (where it is only present in $y\bullet_2$). To track duplication, our security definition can be adjusted to mandate the existence of a bijection between target and source observations. In the present case, constructing the bijection is impossible because there are two target observations *i.e.* $x\bullet_2$ and $y\bullet_2$ in i, leaking the value of y while there is a single source one *i.e.* $y\bullet_2$ in f.

Program j (resp. Program k) leaks the initial value of x through the variable a using a direct (resp. indirect) flow. In either case, the observation $a\bullet_2$ has no counterpart in f and the transformation is rejected. Our Information-Flow preserving definition is semantic in nature and immune to syntactic differences.

```
f: •₁ a = x ^ y;  x = 0; •₂ return a;

g: •₁ a = x ^ y; •₂ return a;
```

Fig. 2. Safe Erasure of One-Time Pad

A. The "Full Information Flow" Paradox

We work with a parametrised attacker model where the attacker makes partial observations. Partial observations are essential, because they capture partial information flows that would go unnoticed if we only had an omniscient attacker observing the whole memory. In the context of dynamic information flow policies, Askarov and Chong [3] have already observed that a program could be secure against a powerful attacker but insecure for a weaker attacker. To see this, consider the program f and the transformed program g in Figure 2. Using a cryptographic analogy, Program f is encoding the plain-text y using the one-time pad key x and erases the key x. Program g performs the same encryption without erasing the key. If the attacker observes $x\bullet_2$ in Program g, he obviously learns the key x. Perhaps surprisingly, the key x can also be learnt for Program f by an attacker observing both the plain-text $y\bullet_2$ and the cipher-text $a\bullet_2$. The attacker obtains x by solving the equation $a\bullet_2 = x\,^\wedge y\bullet_2$ whose unique solution is $x = a\bullet_2{}^\wedge y\bullet_2$. Thus, no additional information flow is introduced and the transformation is (erroneously) deemed secure. To rule out this insecure transformation, we stipulate that both attackers need to observe the same amount of information *i.e.*, the same number n of bits. If both attackers observe a single bit, the observation $a\bullet_2$ in g cannot be simulated in f and therefore we conclude that the transformation is, as expected, not information-flow preserving.

IV. Information-Flow Preservation

A. Execution model

Without loss of generality, we assume that the program state is only made of a bit-addressable memory:

$$\begin{array}{lcl} Mem & = & Addr \to Bit \\ Bit & = & \{0,1\} \end{array}$$

A structured program state s can always be mapped to a memory $m \in Mem$ at the cost of some encoding. For instance, a language using program variables would represent them by reserving certain memory addresses.

The semantics of a program p is given by a one-step deterministic transition relation $\cdot \to_e \cdot \subseteq Mem \times Mem$ where $e \in E = \{\epsilon, o\}$ is either ϵ, denoting a silent transition, or o indicating that the end state of the transition is leaked to the attacker. From an initial memory m^0, a run of a program p is given by a sequence of memories $m^0 \cdot m^1_{e_1} \cdots m^n_{e_n}$ such that for every $i < n$ we have $m^i \to_{e_i} m^{i+1}$. Given such a run, the view of the attacker is given by the sub-trace of leaked memories *i.e.* those memories resulting from a transition tagged o. Formally, we define the transition relation of the attacker

$$\cdot \rightsquigarrow \cdot \subseteq Mem \times Mem$$

as a sequence of silent transitions followed by a transition leaking its final memory:

$$\frac{m \to^*_\epsilon m' \quad m' \to_o m''}{m \rightsquigarrow m''}$$

As a result, from an initial memory m_0, the trace of memories $t = m^1 \cdots m^n$ that is leaked to the attacker is such that for every $i < n$ we have $m^i \rightsquigarrow m^{i+1}$. In the following, we write $(p,c) \Downarrow t$ for a trace of the attacker obtained by running the program p from an initial memory c.

B. Partial Attacker Knowledge

We next define precisely the information leaked through a (partial) trace of attacker observations, using the notion of Attacker Knowledge. Following [5], the Attacker Knowledge from a trace t of program p is defined by

$$\mathcal{K}^t(p) = \{c \in Mem \mid (p,c) \Downarrow t\},$$

i.e., the attacker knowledge corresponds to the initial memories that may lead to (and hence are indistinguishable by) the observation of the trace t. We generalise the notion of Attacker Knowledge to partial observations of n bits of the trace t. A partial observation o is a trace of partial memories, *i.e.*, a sequence of partial maps from addresses to bits:

$$(Addr \hookrightarrow Bit)^*$$

Partial memories $m, m' \in Addr \hookrightarrow Bit$ are ordered so that $m \sqsubseteq m'$ if m and m' agree on all the addresses where m is defined. Formally, this is the standard (point-wise) ordering of partial functions:

$$m \sqsubseteq m' \overset{\triangle}{=} \forall x \in dom(m). m(x) = m'(x).$$

Two partial traces are ordered if they have the same length and the memories (at the same position) are ordered using $\sqsubseteq$. Formally:

$$\frac{}{\epsilon \sqsubseteq \epsilon} \qquad \frac{m \sqsubseteq m' \quad o \sqsubseteq o'}{m \cdot o \sqsubseteq m' \cdot o'}.$$

The number of bits n of a partial trace o is written $|o|$ and is obtained by summing the number of bits defined by the partial memories m in o.

$$\begin{array}{lcl} |\epsilon| & = & 0 \\ |m \cdot o| & = & |dom(m)| + |o|. \end{array}$$

We can then define the notion of partial observation formally as follows.

Definition 1 (Partial Observation)**.** *The set of partial observations of n bits of a trace t is defined by*

$$Obs(t,n) = \{o : (Addr \hookrightarrow Bit)^* \mid o \sqsubseteq t \wedge |o| = n\}$$

Definition 2 generalises the standard notion of Attacker Knowledge to partial observations.

Definition 2. *For a program p and a trace t, the Partial Attacker Knowledge for a partial observation $o \in Obs(t,n)$ is given by*

$$\mathcal{K}^t_n(p,o) = \bigcup_{o \sqsubseteq u} \mathcal{K}^u(p).$$

In Definition 2, we quantify over all the complete traces $u \in Mem^*$ that are compatible with the partial observation o *i.e.*, $\{o \mid o \sqsubseteq u\}$. The Partial Attacker Knowledge is therefore defined as the union of the Attacker Knowledge for all the complete traces that are indistinguishable from the point of view of o.

C. Information-Flow Preserving Transformation

We shall use Partial Attacker Knowledge to define precisely what we mean by a secure transformation of a source program into a target program. Intuitively, a source program p is at least as vulnerable as a target program p' if any partial observation o' of a (target) trace of p' can always be matched by a partial observation o of a (source) trace of p such that the source observation o leaks more information than the target observation o'.

The notion of *matching observation* is formalised by a function mapping partial observations from $Obs(t', n)$ to $Obs(t, n)$. To forbid transformations that increase the lifetime of an information leak, we also enforce that the observations at the target and source level are performed in lock-step *i.e.*, at each instant, both attackers observe the same amount of information.

Definition 3. *The set of lock-step observation mappings Ω is defined by*

$$\Omega(t', t, n) = \{\omega : Obs(t', n) \to Obs(t, n) \mid \forall o', sup(o') = sup(\omega(o'))\}$$

where the support of an observation is defined by

$$sup(\epsilon) = \epsilon \qquad sup(m \cdot o) = |dom(m)| \cdot sup(o)\}.$$

Using lock-step mappings, we are ready to define the security refinement of a source program p by a target program p'.

Definition 4. *A target program p' is at least as secure as p for an attacker observing n bits (written $p \preccurlyeq_n p'$) iff*

$$\forall c, t, t'.(p, c) \Downarrow t \wedge (p', c) \Downarrow t' \Rightarrow \exists \omega \in \Omega(t, t', n). \forall o'. \mathcal{K}_n^t(p, \omega(o')) \subseteq \mathcal{K}_n^{t'}(p', o').$$

Because it quantifies over every partial observations and requires a lock-step mapping, Definition 4 ensures that if a target observation is matched by a source observation, so is every prefix observation.

In order to rule out the duplication of information, we can further restrict the function ω to be injective, by adding the constraint

$$\Omega_1(t', t, n) = Obs(t', n) \rightarrowtail Obs(t, n)$$

where $\rightarrowtail$ denotes the set of bijective functions.

In case of duplication of information, several target observations with the same Partial Attacker Knowledge would need to be mapped to the same source level observation. The fact that ω is injective rules out this possibility. In the following, we take the constraint on matching functions to be either Ω or $\Omega \cup \Omega_1$ and discuss when the constraint Ω_1 is a too strong requirement.

A transformation T is IFP if $T(p)$ is at least as secure as p for attackers with any level of observational power.

Definition 5. *A transformation $T : Prog \to Prog$ is IFP iff*

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

V. A SUFFICIENT CONDITION FOR IFP AND ITS PROOF PRINCIPLE

The definition of an IFP transformation does not suggest an immediate composition proof principle, and proving directly that a program transformation abides to Definition 5 would be a daunting task. To prove that a transformation is IFP, we present a simulation-based proof technique, ensuring that a source memory can be simulated by a target memory.

A. Partition and Simulation Relation

In order to facilitate concrete proofs, we partition the memories leaked to the attacker. In program terms, a typical partitioning would assign to the same partition index i all the memory leaked by a given syntactic observation point $\bullet_i$. To each partition index i, we attach a simulation function $\alpha_i : Addr \to Addr + Bit$. Given a source memory m and a target memory m' mapped to the same partition index i, the mapping α_i explains how m can be effectively simulated by m'. This is done by showing how any bit in m' is either a constant (thus without any information) or a bit that is stored in m possibly at another address as indicated by α_i.

Example 1. *Consider the Programs f and h of Figure 1. For this simple case, each observation point $\bullet_i$ (both in the source and target program) would be mapped to the partition index $i \in \{1, 2\}$. For the initial observation point $\bullet_1$, both memories are the same and therefore α_1 is the identity function $\lambda a.a$. For $\bullet_2$, the memories are the same except for the address x which carry the constant 1 in Program h. Therefore, we would have $\alpha_2 = \lambda a.if\ a = x\ then\ 1\ else\ a$.*

As we show in Theorem 1, these are sufficient conditions to ensure that m' contains at most the same amount of information as m.

The source (resp. target) partition is characterised by a function sid (resp. tid) : $Memory \to [1, \ldots, n]$. Therefore, in the following, we write $m.sid$ (resp. $m'.tid$) for the source index (resp. target index) of a memory m (resp. m').

Definition 6. *Let $\alpha_i : Addr \to Addr + Bit$ be a simulation function indexed by $i \in [1, \ldots, n]$. A source memory m is simulated by a target memory m' (written $m \sim_\alpha m'$) iff we have $m.sid = m'.tid = i$ and*

$$\forall a'. m'(a') = \begin{cases} \alpha_i(a') \text{ if } \alpha_i(a') \in Bit \\ m(\alpha_i(a')) \text{ if } \alpha_i(a') \in Addr \end{cases}$$

Lifted to traces we get,

$$\frac{}{\epsilon \sim_\alpha \epsilon} \qquad \frac{m \sim_\alpha m' \quad t \sim_\alpha t'}{m \cdot t \sim_\alpha m' \cdot t'}$$

In order to avoid duplication of information and enforce the constraint Ω_1, the function α_i needs to be further constrained to forbid mapping distinct target addresses to the same source address *i.e.*, $\exists a \in \mathit{Addr}. a = \alpha_i(a_1) = \alpha_i(a_2) \Rightarrow a_1 = a_2$.

Using the previous definitions, our necessary conditions for IFP can be stated using Theorem 1.

Theorem 1. *Consider two programs p and p', and an indexed simulation function $\alpha_i : \mathit{Addr} \to \mathit{Addr} + \mathit{Bit}$. If*

$$\forall c, t, t'. \begin{pmatrix} (p, c) \Downarrow t \\ \wedge \\ (p', c) \Downarrow t' \end{pmatrix} \Rightarrow t \sim_\alpha t'$$

then $\forall n, p \preccurlyeq_n p'$, i.e.*, the transformation is IFP.*

Proof: Suppose $t \sim_\alpha t'$. We need to prove, for any n,

$$\exists \omega. \forall o'. \quad \mathcal{K}_n^t(p, \omega(o')) \subseteq \mathcal{K}_n^{t'}(p', o').$$

Let $t' = m'_0 \cdots \cdots m'_n$ and $t = m_0 \cdots \cdots m_n$. Remember that ω has type $\mathit{Obs}(t', n) \to \mathit{Obs}(t, n)$ and that observations are made in lock-step fashion. As a result, it is sufficient to provide a mapping $\omega_i : \mathit{Obst}(m'_i, k) \to \mathit{Obs}(m_i, k)$ which given a partial observation o' of the target memory m'_i reconstructs an observation $\omega_i(o')$ of the source memory m_i. As $m_i \sim_{\alpha_i} m'_i$, this can be done as follows. Suppose that $o'(a') \neq \perp$ for some address a'. If $\alpha_i(a') \in \mathit{Addr}$, we can obtain the same observation from memory m_i and $\omega_i(o')(a') = m_i(\alpha(a'))$. Otherwise, If $\alpha_i(a') \in \mathit{Bit}$, the observation does not provide any information and therefore it suffices to pick an arbitrary fresh address a, and have $\omega_i(o')(a) = m_i(a)$.

From this construction, for any partial observation $o' \in \mathit{Obs}(t', n)$ we can derive a partial observation of the source execution $\omega(o')$ which contains at least as much information as in o'. It follows that the target attacker knowledge obtained from an observation o' is always bigger than the source attacker knowledge of the observation $\omega(o')$ and therefore the property holds.

■

Theorem 1 does not provide a complete characterisation of IFP transformations. It captures transformations where information is perhaps moved around but is otherwise unmodified. To illustrate the limitation, consider the following contrived example

$$x{:=}y \quad \to \quad x'{:=} \sim y$$

where the bit values of y are flipped bitwise. The variables x and x' contain the exact same information *i.e.*, the value of y. Yet, the transformation of values cannot be modelled by a simulation function α. We show that standard optimisations including register allocation and dead store elimination are in the scope of Theorem 1.

B. Simulation-Based Principle

The pre-condition of Theorem 1 can be proved using a lock-step backward simulation principle over the attacker semantics.

Definition 7 (Backward Simulation)**.** *A simulation function $\alpha_i : \mathit{Addr} \to \mathit{Addr} + \mathit{Bit}$ is a backward simulation if:*

1) *From the initial memory c, the first source memory m and target memory m' leaked to the attacker are in relation ($m \sim_\alpha m'$).*
2) *Given memories $m_1 \sim_\alpha m'_1$, for every attacker step of the target program $m'_1 \rightsquigarrow m'_2$ there exists a memory m_2 in the source program such that $m_1 \rightsquigarrow m_2$ and $m_2 \sim_\alpha m'_2$.*

Theorem 2. *Suppose two programs p and p' and a simulation function α. If there is a backward simulation (according to Definition 7) for α, then the pre-condition of Theorem 1 holds* i.e.*,*

$$\forall c, t, t'. \begin{pmatrix} (p, c) \Downarrow t \\ \wedge \\ (p', c) \Downarrow t' \end{pmatrix} \Rightarrow t \sim_\alpha t'.$$

Proof: Thee proof is by induction over the length of the trace t' and follows using Condition 1 of Definition 7 for the base case and Condition 2 of Definition 7 for the inductive case.

■

VI. Translation Validation for Register Allocation

Translation validation [24] is a verification technique which consists in validating *a posteriori* (and automatically) that a program p' is obtained by a valid transformation of a program p. We adapt this principle and design a specialised algorithm to validate *a posteriori* whether programs obtained through the Register Allocation (RA) pass of the verified CompCert C compiler [20] satisfy our IFP property. This is done by explicitly constructing a backward simulation using the sufficient condition of Section V. An interesting feature of our algorithm is that certain failures can be interpreted as potential information leak and that these leaks can be closed automatically by inserting erasing instructions.

A. Register Allocation in a Nutshell

Before RA, programs make use of an unbounded number of pseudo-registers. The role of RA is to explicit the constraint that the physical machine has only finitely many registers and therefore allocate each pseudo-register to a machine register. RA is also responsible for implementing calling-conventions *i.e.* passing arguments in the right register and restoring *callee-saved* registers. What makes RA a complex optimisation task is that this resource allocation task may be impossible due to a shortage of registers. In that case, a register may be *spilled* in the function stack frame *i.e.* its content copied, for later reuse. After the last use of a spilled register, a conventional RA algorithm has no reason to explicitly erase the stack location. This breaks our IFP property because i) the value is duplicated (it is stored in both a register and a stack location); ii) this introduces an information leak if the stack location is not erased after the last use of the register. This is illustrated by Example 2.

Example 2. *Consider the simple function* cipher *of Figure 3 and the function* cipher' *obtained by a typical RA pass for an hypothetical target architecture with only two registers r1 and r2. The code has been arranged to highlight the relation*

```
1 cipher(text)
2  key   := get_key();
3  salt  := get_salt();
4
5
6  tmp   := text^salt;
7
8  key   := key^tmp;
9  return(key)  •
```

```
1 cipher'(text')
2  r2    := get_key();
3  r1    := get_salt();
4  key' := r2;          // spill
5  r2    := text';      // load
6  r2    := r2^r1;
7  r1    := key'        // load
8  r1    := r1^r2;
9  return;  •
```

Fig. 3. Original program (left) After register allocation (right)

between a source instruction and the corresponding sequence of target instructions. For instance, the source instruction of Line 6 is compiled into the sequence of target instructions of Lines 4-6. The Function cipher' contains additional spilling/loading instructions inserted by RA Lines 4,5 and 7. In Function cipher the secret value in key is overwritten Line 8, hence an attacker at • cannot observe its value. However in Function cipher', the value of key in stored in register r2 and copied in variable key' (see Line 4). As the variable key' is never erased, an attacker at • can observe its value. Hence this transformation is not IFP and will be rejected by our validation algorithm.

B. Register Allocation in CompCert

1) Register Allocation Languages: In CompCert, the RA pass compiles a source in the Register Transfer Language (RTL) into a target in the Location Transfer Language (LTL). RTL and LTL are both fairly classic control-flow graph program representations where nodes are labelled with three-address code instructions.

Instructions representative of RTL are given below.

$$\begin{array}{rcl} I \ni i & ::= & \texttt{nop}(s) \\ & | & \texttt{op}(op, \overline{args}, dest, s) \\ & | & \texttt{load}(addr, \overline{args}, dest, s) \\ & | & \texttt{store}(addr, \overline{args}, src, s) \\ & | & \texttt{cond}(cond, \overline{args}, s_{true}, s_{false}) \\ & | & \texttt{call}(sig, fid, \overline{args}, dest, s) \\ & | & \texttt{return}(arg) \end{array}$$

Each instruction $i \in I$, attached to a node $n \in \mathbb{N}$, specifies its immediate successor s which is the node of the instruction to execute next. The `nop` instruction does nothing. The `op` instruction computes the value of the variable $dest$ using the values stored in the vector of pseudo-registers $\overline{args}$. The `load` instruction moves the value at the address computed by $\overline{args}$ with the addressing mode $addr$ to the destination $dest$. Similarly `store` move the value from src to the address computed by $\overline{args}$ and $addr$. The `call` instruction calls the function fid with signature sig with parameters $\overline{args}$; the result of the call is stored in $dest$. The instruction `cond` models a conditional and has two successors. Depending on the value of arg, the instruction to execute next is either s_{true} of s_{false}. Finally, `return` exits the current function and return the value of arg.

The LTL language is similar to RTL with the differences that i) so-called locations corresponding to stack slots or machine registers are used in place of the unlimited pseudo-registers called temporaries and; ii) `call` and `return` instructions are bounded by the calling conventions of the target architecture. As a result, in the program syntax, $\overline{args}$ and $dest$ are locations made of either machine registers or stack slots used for spilling registers. Compared to temporaries, stack slots are pointers into the current stack frame and special care must be taken to make sure that spilled registers do not overlap. The calling conventions stipulate where function arguments and return must be stored and are represented by a list of locations. For instance, for x86_32, arguments are passed on the stack and the result is stored in eax while for x86_64, arguments are passed in different registers depending of the type of the argument.

2) Translation Validation of RA : CompCert is using an untrusted RA algorithm whose output is verified using a specialised translation validation approach [25], ensuring that each target LTL function is a sound compilation of the source RTL function. Observational correctness of the whole program is then achieved by composing the validation of each pair of functions. As our own IFP validator is reusing some key components, we give a brief overview of their algorithm.

The translation validator of CompCert exploits that the RTL and LTL functions have a very similar structure and only differ in that the LTL function introduces *move* instructions to materialise spills and reloads of stack slots; and data movements between registers.

The untrusted RA algorithm takes as argument an RTL function and returns an LTL function. The LTL function is structured in such a way that it is straightforward to rebuild a mapping from a single RTL instruction to the list of LTL instructions it is compiled into. Because RA is only using a few local transformations, the list of LTL instructions is always made of *move* instructions *i.e.* assignments between locations, followed by the actual instruction which is obtained from the original RTL instruction by replacing registers by locations. The only possibilities are presented in the first column of Figure 4. Each possible association of instructions are categorized into block-shape and form whole block-shape functions. Similar to RTL and LTL, block-shape functions are structured as a CFG and each block-shape is parametrised with the nodes of its successors. Moreover each possible block-shape contains a list of move instructions from the LTL code, labelled mv in Figure 4, which are executed before the core instruction.

COMPCERT translation validation algorithm is composed of two parts. The first one is a structural check which verifies that the LTL function respects a certain structure with respect to the RTL function. This check is carried out while constructing the block-shape function. For example, to construct `BSop` the validator needs to find `op` instructions in both RTL and LTL and check that the operations match. Another example is `BScond` where the condition and successors must be the same in both `cond` instructions. If an unexpected pattern is found between the source and transformed functions then the structural check fails and so does the validation. To complete the translation validation algorithm, COMPCERT performs an additional backward data flow analysis to verify that the two functions effectively compute and use the same values.

C. Modular IFP Validation Algorithm

Our security policy is determined by the location of observation points $\bullet$ in both the RTL and LTL programs. In order to get a translation validation algorithm integrated in the RA compiler pass, it is necessary to be able to process one function at a time. Our solution is to set observation points at function boundaries, more precisely at function calls and returns. From a security point of view, this ensures that the LTL locations do not leak information at function return *i.e.* the stack frame of LTL only contains information that is also present in the pseudo-registers of RTL.

In the following, we detail our IFP validation algorithm. In Section V, we have shown that proving IFP preservation can be done by exhibiting some mapping $\alpha_i : \mathit{Addr} \to \mathit{Addr}+\mathit{Bit}$ used to establish a simulation between the source and the target memories at every observation point $\bullet_i$. To get to this point, our IFP validator needs to construct richer objects in order to cater for the RTL and LTL COMPCERT memory model [22] and the fact that, for intermediate program points, the existence of such an α_i mapping is too strong a requirement. Hence, the IFP validator constructs a set of associations between locations and temporaries $\beta_i \in \mathcal{P}(\mathit{Loc} \times \mathit{Temp})$; computes the set of modified location $\gamma_i \in \mathcal{P}(\mathit{Loc})$ and performs a constant analysis $\mathit{cst}_i : \mathit{Loc} \to \mathit{Value}+\{\top\}$ such that, given a program point i,

- $(l, t) \in \beta_i$ iff the value of the LTL location l is the same value as the RTL temporary t,
- $l \in \gamma_i$ iff the location l may be modified by the current function,
- $\mathit{cst}_i(l) = v$ iff for any execution the location l always contains the value v.

The constant analysis cst_i and the set of modified locations γ_i are computed by iteratively solving standard forward data-flow equations using the existing data-flow framework of COMPCERT.

1) Inference of Location Mapping β_i: Compared to the existing RA validator of COMPCERT, a difference is that we construct β_i using a forward data-flow analysis over the block-shapes of Figure 4. The reason is that, for compiler correctness, it is just necessary to ensure that the return LTL register, say eax, is mapped to the return RTL temporary, say t. For IFP, we need a complete mapping for all the RTL temporaries and LTL locations.

The transfer functions for the possible block shapes generated by RA can be found in Figure 4. The transfer functions are using the following notations. We write $(l \to t)$ for a pair $(l, t) \in \beta_i$ with the interpretation that the LTL location l is mapped to the RTL temporary t. We extend this notation to vectors of locations and temporaries and write $(l_1, \ldots, l_n) \to (t_1, \ldots, t_n)$ for the set $\{(l_1 \to t_1), \ldots, (l_n \to t_n)\}$. Because of copy instructions, in both LTL and RTL, the mapping is not unique and there may be several pairs with l as first component and t as the second component. Given a temporary t, we write $(\_ \to t)$ for the set of all pairs such that the second element is t. Symmetrically, for a given location, we write $(l \to \_)$ for the set of all pairs such that the first element is l.

All block shapes have a sequence of move mv instructions on the LTL side. A move $l_1 = l_2$ assigns l_2 to l_1. Given an initial mapping B, the effect of move $l_1 = l_2$ is to remove the existing mappings for l_1 $((l_1 \mapsto \_))$ and map l_1 to existing mappings of l_2 in B. As a result, we get

$$\mathit{transfer}_{l_1 = l_2} = B \setminus (l_1 \to \_) \cup (\{l_1 \to x\} \mid \{l_2 \to x\} \in B).$$

For valid code, there should always be an existing mapping for l_2. If not, the analysis continues but l_2 and l_1 would be flagged as potential information leaks at the next observation point. A `BSnop` only consists in a list of LTL move instruction and its effect is modelled by iterating the transfer function for a single move instruction. For `BSop`, we check after computing the consequence of the moves that the LTL arguments $\overline{args'}$ are mapped to the RTL arguments $\overline{args}$. If this is the case, we have the guarantee that the arguments $\overline{args}$ and $\overline{args'}$ have the same values and therefore the destinations tmp and loc also have the same value. After evaluating the effect of the moves, the transfer function invalidates the existing mappings for tmp and loc and adds the mapping $(\mathit{loc} \to \mathit{tmp})$. For `BSLoad`, the transfer function is similar but exploits the additional invariant that the memory of RTL and LTL agree for every address a that is neither an LTL location $a \notin \mathit{Loc}$ nor a temporary $a \notin \mathit{Temp}$. For `BSstore`, we check that the computed addresses $\overline{args}$ and $\overline{args'}$ compute the same value in both RTL and LTL. We also check that the stored value are the same *i.e.*, the current mapping includes $(\mathit{loc} \to \mathit{tmp})$. Except for the potential move instructions, a memory store has no effect on the current mapping. Yet, our verifications ensure that the LTL and RTL memory still agree for addresses that are neither temporaries nor locations. For `BScall`, we check that the arguments of the call are the same. In RTL, the arguments and return value are explicitly passed; in LTL, the functions $\mathit{Params}(\mathit{sig})$ and $\mathit{Ret}(\mathit{sig})$ implement the architecture dependent calling conventions of

Block Shape	*Conditions*	*Transfer Function*
$\texttt{BSnop}(s)$: $\texttt{nop};$ $\Big\vert$ $mv;$ $\texttt{nop};$		$transfer_{mv}(mv, B)$
$\texttt{BSop}(s)$: $tmp := \texttt{op}(\overline{args});$ $\Big\vert$ $mv;$ $loc := \texttt{op}(\overline{args'});$	$(\overline{args'} \to \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$ $\setminus (\_ \to tmp) \setminus (loc \to \_)$ $\cup \{loc \to tmp\}$
$\texttt{BSload}(s)$: $tmp := [addr(\overline{args})];$ $\Big\vert$ $mv;$ $loc := [addr(\overline{args'})];$	$(\overline{args'} \to \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$ $\setminus (\_ \to tmp) \setminus (loc \to \_)$ $\cup \{loc \to tmp\}$
$\texttt{BSstore}(s)$: $[addr(\overline{args})] := tmp;$ $\Big\vert$ $mv;$ $[addr(\overline{args'})] := loc;$	$(loc \to tmp) \in transfer_{mv}(B) \;\wedge$ $(\overline{args'} \to \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$
$\texttt{BScall}(s)$: $tmp := f(sig, args)$ $\Big\vert$ $mv_1;$ $Ret(sig) := f();$ $mv_2;$	$(\overline{Params(sig)} \to \overline{args})$ $\subseteq \; transfer_{mv_1}(B)$	$transfer_{mv}(mv_2,$ $\{Ret(sig) \to tmp\} \cup$ $callee\text{-}save(transfer_{mv}(mv_1, B)))$
$\texttt{BScond}(s1, s2)$: $\texttt{cond}(cond, \overline{args})$; $\Big\vert$ $mv;$ $\texttt{cond}(cond, \overline{args'});$	$(\overline{args'} \to \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$
$\texttt{BSreturn}()$: $\texttt{ret}(arg);$ $\Big\vert$ $mv;$ $\texttt{ret};$	$(arg' \to arg) \in transfer_{mv}(B)$	$transfer_{mv}(mv, B)$

Fig. 4. Transfer functions for β_i

functions arguments and return. In RTL, all the temporaries are restored after a function call. In LTL, the calling conventions state that both stack locations and a subset of the registers *i.e.*, so called *callee-saved* registers, are restored after a call. This is modelled in the transfer function by the function *callee-save* which invalidates mappings to registers which are not callee-saved. For BScond, we simply check that the conditions evaluate to the same value thus ensuring that both program have the same control-flow. For BSreturn, we also check that the return values are the same and only update the mapping to model move instructions.

In order to get β_i, we iterate the data-flow equations until a fixpoint using data-flow framework of COMPCERT.

2) Verification of Sufficient Conditions: If the computation of β_i succeeds, the next step consists in verifying, for every observation point, the IFP verification conditions. For a given observation point $\bullet_i$, we verify that for every LTL location $l \in \gamma_i$ that may be modified by the current function, there exists either a mapping to an RTL register t *i.e.*, $(l \to t) \in \beta_i$ or the location l is constant $cst_i(l) = v$ for some $v \neq \top$.

Theorem 3. *Let p be an RTL program and p' be an LTL program. Suppose that we have successfully constructed β_i, γ_i and cst_i for every program point of p. If for every observation point $\bullet_i$, the following condition hold:*

$$\forall l \in \gamma_i.(\exists t.(l \to t) \in \beta_i) \vee (\exists v.cst_i(l) = v \wedge v \neq \top)$$

then the RA transformation of p to p' is IFP.

Moreover, if every pair (l_1, l_2) of distinct locations is mapped to distinct temporaries, the transformation prevents data duplication.

In the following section, we give some key insights on how the previous verification conditions are sufficient to ensure the existence of a simulation function α_i and a backward simulation between p and p'.

D. Correctness of the Validation Algorithm

We suppose an execution of p and p' where memories m_1 and m'_1 are respectively leaked from observation point $\bullet_1$ and there exists α_1 such that $m_1 \sim_{\alpha_1} m'_1$. Similarly we note $\bullet_2$ the observation point leaking m_2. It remains to prove that we can construct a backward simulation according to Definition 7. Hence we need to prove two goals: the existence of m_2 and α_2 such that $m_1 \rightsquigarrow m_2$ and $m_2 \sim_{\alpha_2} m_2'$.

1) $\exists m_2,\ m_1 \rightsquigarrow m_2$: By definition of $\rightsquigarrow$, there is a derivation of length n' such that $m_1' \rightarrow^{n'} m_2'$. We have to show that there is a derivation of length n such that $m_1 \rightarrow^{n} m_2$. Such a proof is already needed by the original COMPCERT semantics preservation proof of the RA pass [25] and we therefore inherit it for free.

2) $\exists \alpha_2,\ m_2 \sim_{\alpha_2} m_2'$: The construction of α_2 partitions addresses in the following three categories:

i) addresses which are not LTL locations;
ii) LTL locations that may be modified by the current function;
iii) and LTL locations that have not been modified.

First, consider addresses that are not locations in the LTL memory model . The memory content at these addresses can only be modified via `load` and `store` instructions. While building β_2 we check that these memory accesses always agree on their computes arguments. Therefore in p and p', the memory content remains the same and α_2 is the identity function for these addresses.

Second, consider LTL locations that may have been modified since the start of the current function *i.e.* $l \in \gamma_2$. We know from our precondition of Theorem 3 that we are able to map l to either a temporary or a constant. Thus α_2 is equal to β_2 or cst_2 for every location of $l \in \gamma_2$.

Third, consider for LTL locations that are necessarily unmodified *i.e.* $l \notin \gamma_2$. We know that when reaching $\bullet_1$ the memories m_1 and m_1' were leaked and that $m_1 \sim_{\alpha_1} m_1'$. Moreover locations and temporaries are local to their functions and observation points are placed before function calls and returns. From these facts we deduce that $\alpha_1(l)$ still holds at $\bullet_2$ and gives us $\forall (l \notin \gamma_2),\ \alpha_2(l) = \alpha_1(l)$.

Lastly, to ensure the absence of duplication of information, we need to prove that α_2 is injective. For addresses which are not locations, α_2 is the identity function which is injective. For addresses not in γ_2 this is given by the fact that by induction α_1 is also injective. For the other addresses we have the hypothesis that every location maps to distinct temporaries which proves that α_2 is injective.

E. Validating Dead Store Elimination (DSE)

DSE is an optimisation which replaces an assignment to RTL temporary *tmp* by `nop` when the compiler is able to show that the value of *tmp* will not be used by the rest of the computation. When such instructions have the security purpose of erasing sensitive values from memory, DSE is introducing information-flow leaks. A particularity of COMPCERT is that dead store elimination is not a separate pass but is performed during RA. The rationale is that a liveness analysis is needed to both detect *dead store* and construct the interference graph of RA algorithms [15]. Therefore the validator of COMPCERT is capable of validating both RA and DSE. Similarly, we also adapt our IFP validator to handle DSE. For most cases, DSE will not be IFP. To cope with this situation, we refer to Section VI-F where we show how to patch the program on the fly to close potential information leaks. A typical case where removing a store is not IFP occurs when it happens just before an observation point.

The only transformation that may occur in DSE is to replace an assignment by a `nop`. To deal with this situation, our validator considers the additional block shapes and transfer functions of Figure 5. Those blocks are `BSopdead` and `BSloaddead` where the corresponding core instructions in LTL has become a `nop`. In both cases, after evaluating the `move` instructions, the transfer functions remove the mappings for the target RTL temporary *tmp*.

With these extensions our IFP validator is now able to check the COMPCERT transformation composed of register allocation and DSE.

F. Patching algorithm

An interesting property of our IFP validator is that we can recover from certain validation failures; track down the origin of the information leak and apply a patch to a function f' to automatically close the information leak. When this is the case, it is possible to run existing optimisations unmodified, and during a post-treatment, detect and remove potential information leaks. Suppose that, for a given observation point $\bullet_i$, the verification conditions needed by Theorem 3 do not hold. Typically, there is a location l that is neither a constant ($cst_i = \top$) nor has a mapping to some RTL temporary ($\beta_i(l) = \emptyset$). In this situation either the validator is not precise enough or l is responsible for an information leak. To close the potential leak detected by the IFP validator, our patching algorithm inserts an erasure instruction and sets the location to the constant 0 just before the observation point $\bullet_i$. After the addition of those erasure instructions, we have the guarantee that the transformed program is IFP. Moreover, those erasure instructions cannot compromise the correctness of the transformation. The rationale is that an erasure instruction for location l is necessarily a *dead* store. To see this, consider by contradiction that l is live. In that case, to establish semantics preservation, the original COMPCERT validator needs to establish a mapping for location l and an RTL temporary t. As our forward validator always computes more mappings than the original backward validator of COMPCERT, it would establish the same mapping. This contradicts our assumption.

As a result, we have the guarantee that inserting erasure instructions, according to the rules above, is a sound algorithm to make COMPCERT RA a semantic preserving IFP transformation.

VII. EXPERIMENTS

The translation validation algorithm of Section VI has been integrated as part of the register allocation pass of COMPCERT [20]. We run our IFP validator; close any potential information leak using the patching algorithm of Section VI-F. Afterwards, we run the existing validator of COMPCERT, thus, ensuring the semantics correctness of our security transformation.

Block Shape	*Conditions*	*Transfer Function*
$\texttt{BSopdead}(s)$: $tmp := \texttt{op}(\overline{args});$ $\left\vert \begin{array}{l} mv; \\ \texttt{nop}; \end{array}\right.$		$transfer_{mv}(mv, B) \setminus (\_ \to tmp)$
$\texttt{BSloaddead}(s)$: $tmp := [addr(\overline{args})];$ $\left\vert \begin{array}{l} mv; \\ \texttt{nop}; \end{array}\right.$		$transfer_{mv}(mv, B) \setminus (\_ \to tmp)$

Fig. 5. Additional transfer functions for DSE

In our model, the attacker can only observe the memory at so-called observation points. As explain in Section VI, observation points are set at function boundaries: the attacker may observe memory just before call and return instructions. With this policy, our IFP validator enforces that register allocation does not introduce information leaks due, for instance, to stack allocated variables (or spilled registers) not being properly erased at function return; an acknowledged security issue [14], [28], [32].

A. Results and analysis

The experiments have been conducted on a quad-core Intel i7-6600U at 2.60GHz with 16GB of RAM running Fedora 27. We have tested our IFP validator on 24 programs which are all part of the CompCert test suite. For every program, our validator detected potential information leaks introduced by CompCert RA. All the information leaks have been successfully closed by our patching algorithm and all the resulting programs have passed the CompCert validator.

The impact of the security transformation on the size of the programs and their efficiency are summarised in Figure 6. The first bar represents the runtime overhead of our secured RA compared to the original RA of CompCert. The benchmarks have a running time ranging from 1 to 10 seconds and are sorted by increasing overhead that is obtained by averaging 50 runs. The second (grey) bar is the overhead on the size of the program incurred by the insertion of erasure instructions. The third (dark) bar represents the overhead in the number of executed instruction.

As shown in Figure 6, the increase in program size can be large and is above 20% for most of the benchmarks. Yet, this increase is not always reflected in the execution overhead because even when many erasure instructions are inserted, they may be rarely executed. Actually, for most functions, the increase of the number of executed instructions is negligible and for more than half of the programs the execution overhead is within a range of $\pm 5\%$. We believe that speed ups are a lucky side-effect, probably due to an improved behaviour of the cache of instructions. Yet, there 4 benchmarks for which the execution overhead is above 10%. For `chomp`, the overhead peaks at 51% but is not quite correlated with the increase in executed instructions which is less than 15%. Another extreme phenomenon is the speedup of 17% of `lists` for which there is no obvious explanation

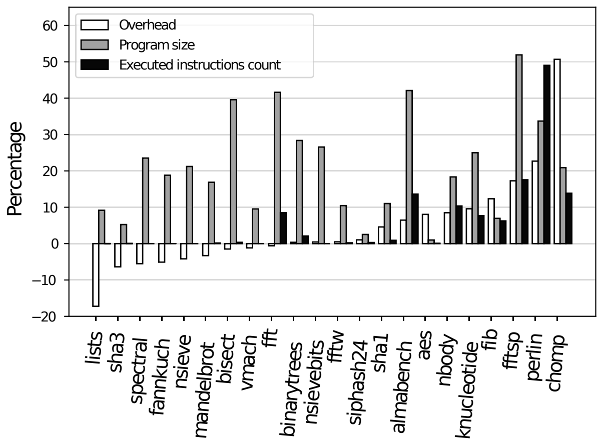

Fig. 6. Patched programs compared to original compared programs

and require further investigations. Anyway, these results are encouraging and show that an improved security can often be obtained without sacrificing too much efficiency.

For a few benchmarks, the number of inserted instructions can be quite large. For instance, for programs like *almabench* or *fftsp* we notice a 40% increase in program size. A reason for this phenomenon is that our IFP validator needs to be quite conservative about the behaviour of function calls relying exclusively on guarantees given by calling conventions. This is illustrated by Example 3 where a potentially spurious instruction needs to be inserted to protect against a potential information leak.

Example 3. *Consider the code snippet of Figure 7.*

```
r := g(x);                eax := g(edi);
x := 2;                   eax := 2;
return x;                 return;
```

Fig. 7. RTL program (left), LTL program (right)

According to standard x86 calling conventions, it is typically compiled as the LTL program of Figure 7. After the call of g*, we only have the guarantee that registers are either overwritten or carry the same value as before. Therefore, our validator makes the conservative assumption that* edi *(that is* *not callee-saved) was not modified by g, may leak the initial value of the variable x and therefore needs to be explicitly erased, thus adding a potential spurious erasure instruction* $edi := 0$.

As programs like *almabench* or *fftsp* are using a significant number of distinct registers, the patching algorithm introduces for each of them an erasing instruction for each call site.

B. Discussion

The validator is also verifying whether register allocation satisfies the absence of data duplication. This property is unfortunately currently violated and cannot be easily fixed because of constraints imposed by existing calling conventions. More precisely, calling conventions stipulate that arguments are copied into specific registers. Moreover, as arguments are not callee-saved, erasing the original register would be unsound and, therefore, this duplication is inevitable. At the price of breaking compatibility with existing code, a solution would be to adapt the calling-conventions so that arguments are callee-saved. Other sources of duplication are between a register and a spilled stack slot. In this case, either the register or the stack slot is dead and could be erased. The patching algorithm would have to be enhanced with a liveness analysis in order to perform this reasoning and erase the dead location.

Reducing the number of erasure instructions would require to give the validator additional information about the registers that are either preserved or erased by function calls. For functions in the same compilation unit, this could be done using standard static analyses computing unmodified registers or constant registers. Library calls are more problematic. A possibility would be to require extended calling conventions stipulating which registers are preserved (beyond callee-saved) or erased. However, if the client code is optimised for these extended calling conventions, it would only be secure if linked against this specific version of the library.

VIII. Securing other Compiler Passes

In their paper, D'Silva *et al.* [14] mention that compiler passes such as *Code Motion* an *Inlining*, because they modify the lifetime of variables, may introduce information leaks. In the following, we explain how these transformations fit with our IFP property.

A. Inlining

In our experiments, observation points are attached to function calls. As inlining consists in replacing a function call by a function body, if has the effect of removing an observation point. As observations need to be synchronised, inlining breaks this property and therefore is simply not an IFP transformation. As this may seem overly restrictive, a first solution to accommodate some inlining would be to weaken the security policy and attach observation points only to security critical functions. As a result, those critical functions would not be inlined but other functions could be freely inlined without modifying observation points. In this restricted scenario, inlining would be an IFP transformation. Another tempting approach would be to detach the observation point from the function call. Yet, this raises the issue of code motion that we discuss below.

B. Code motion

The risk of code motion is that code, and therefore information, could move around observation points and thus modify the security policy. Consider for instance, the code of Figure 8 where observation points are detached from function calls and the code is subject to code motion. In this example, both

```
                         x := f();
•1 x := f();             y := g();
x := 0;                  x := 0;
•2 y := g();             y := 0;
y := 0;                  •1•2
```

Fig. 8. Code motion and detached observation points

erasure instructions and observation points have been delayed and moved at the end of the program. The result is that the transformation is formally IFP but defeats the intention of the security policy. It follows that a IFP-aware compiler needs to limit code motion within the bounds of observation points.

IX. Related work

The *correctness-security* gap was pinpointed by D'Silva *et al.* [14]. In that work, dead-store elimination is identified as a program transformation that can cause security issues. Deng and Namjoshi [11]–[13] introduce an information flow notion of *leaky triple* and ensure that an adaptation of the standard DSE transformation [13] and SSA transform [12] preserves this notion of information flow. Leaky triples do not capture the amount of information that is leaked. As a consequence, they ensure that a non-interferent program is transformed into another non-interferent program but a leaky program may be transformed into a more leaky program without violating their property. Because our IFP property is based on the notion of partial attacker knowledge, it will rule out this possibility and ensure that the transformed program is no more leaky.

Chong and Myers [9] propose semantics foundations for defining erasure policies. A main insight behind that work is that erasure can be seen as the dual of declassification [26]. Later on, Chong and Myers [10], but also Hunt and Sands [16], propose type-systems for verifying erasure policies of the source code. Askarov *et al.* [4] 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. We believe that our IFP definition has the potential to preserve such rich erasure policies. A difficulty is to specify the observation points relevant to the source property and make sure that they are preserved across compiler passes. An attacker observing all the intermediate memory states would probably suffice but may preclude too many transformations.

Yang *et al.* [32] provides a list of techniques used by developers to prevent compilers from removing security-sensitive dead-store instructions. They present a secure dead-store elimination transformation that has been implemented for LLVM. This transformation verifies our IFP property. Simon, Chisnall and Anderson [29] investigate how compilers may break the security of cryptographic code because it is impossible to control side-effects of compiler optimizations. They propose compiler support for constant-time selection and for the secure erasure of secrets, which improves the security of the generated code.

Besson *et al.* [7] pursue a goal similar to ours, *i.e.*, preserving information flow under program optimisations. The attacker in [7] is weaker because the attacker can only observe memory at the end of the execution. By basing our attacker model on traces of observations, we obtain a model that is more realistic and closer to the intuitive notion of an attacker that probes memory at specific moments during execution. From a formal point of view, the move to a trace-based model is necessary to analyse the security of intra-procedural optimisations. In addition, we show how it is possible to combine identification and patching of leaky transformations with a solution to the problem of identifying leaks due to the duplication of data. Finally, contrary to [7], we have experimentally validated our technique by analysing and enforcing information preservation properties of passes of a realistic compiler, the CompCert C compiler.

Barthe *et al.* [6] propose a general framework for reasoning about the preservation of information leakage. Their approach is based on an instrumented semantics which explicitly leaks information to the attacker and which is used for deriving reasoning principles based on so-called 2-simulations. As our attacker may perform arbitrary observations of memory, we quantify over a set of leakage functions. Moreover, our reasoning principle is simpler and is sufficient to validate most transformations. Yet, it also has limitations and 2-simulations are more flexible when some leaked information of the source is gradually leaked in the target program.

X. Conclusion

We present a notion of Information-Flow Preservation ensuring that an attacker observing n bits of information during the run of a target program does not get an advantage over an attacker observing the same amount of bits of the source program. This is a strong property which is not tailored to a specific model of leakage but ensures that a compiler does not introduce any kind of information-flow leaks. We show that it is possible to adapt classic aggressive optimisations to this setting by inserting automatically erasing instructions before observation points. This approach has the advantage that the existing compiler code can be reused mostly unchanged. Our information-flow preservation property may prevent the spatial duplication of information using an injection between the observations of the target and source attackers. As future work, we will investigate how to characterise in the same framework both spatial and temporal duplication of data, thus modelling the lifetime of data as temporal duplication. The notion of injection is unfit for this purpose and would need to be generalised to allow up-to k duplications of the same data during the program execution. We also intend to push our security property to its limit and model an attacker observing memory at any time. Though the opportunities for optimisations will most probably be greatly reduced, we are nonetheless confident that programs can still be compiled in a reasonable manner under this stringent requirement, at the cost of some budget for temporal duplication of data.

Acknowledgement: This work has been partially funded by the French ANR project Scrypt ANR-18-CE25-0014.

References

[1] "HOL4," http://hol.sourceforge.net.

[2] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security 16*. USENIX, 2016, pp. 53–70.

[3] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *CSF 2012*. IEEE Computer Society, 2012, pp. 308–322.

[4] A. Askarov, S. Moore, C. Dimoulas, and S. Chong, "Cryptographic Enforcement of Language-Based Information Erasure," in *CSF'15*. IEEE, 2015, pp. 334–348. [Online]. Available: https://doi.org/10.1109/CSF.2015.30

[5] A. Askarov and A. Sabelfeld, "Gradual Release: Unifying Declassification, Encryption and Key Release Policies," in *SP'07*. IEEE, 2007, pp. 207–221. [Online]. Available: https://doi.org/10.1109/SP.2007.22

[6] G. Barthe, B. Grégoire, and V. Laporte, "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"," in *CSF 2018*. IEEE, 2018, pp. 328–343.

[7] F. Besson, A. Dang, and T. Jensen, "Securing Compilation Against Memory Probing," in *PLAS'18*. ACM, 2018, pp. 29–40.

[8] S. Blazy, D. Pichardie, and A. Trieu, "Verifying Constant-Time Implementations by Abstract Interpretation," in *ESORICS 2017*, ser. LNCS, vol. 10492. Springer, 2017, pp. 260–277.

[9] S. Chong and A. C. Myers, "Language-Based Information Erasure," in *CSFW'05*. IEEE, 2005, pp. 241–254. [Online]. Available: https://doi.org/10.1109/CSFW.2005.19

[10] ——, "End-to-End Enforcement of Erasure and Declassification," in *CSF'08*. IEEE, 2008, pp. 98–111. [Online]. Available: https://doi.org/10.1109/CSF.2008.12

[11] C. Deng and K. S. Namjoshi, "Securing a Compiler Transformation," in *SAS*, ser. LNCS, vol. 9837. Springer, 2016, pp. 170–188.

[12] ——, "Securing the SSA transform," in *SAS*, ser. LNCS, vol. 10422. Springer, 2017, pp. 88–105.

[13] ——, "Securing a compiler transformation," *Formal Methods in System Design*, vol. 53, no. 2, pp. 166–188, 2018. [Online]. Available: https://doi.org/10.1007/s10703-017-0313-8

[14] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *SPW '15*. IEEE, 2015, pp. 73–87. [Online]. Available: http://dx.doi.org/10.1109/SPW.2015.33

[15] L. George and A. W. Appel, "Iterated register coalescing," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 300–324, 1996. [Online]. Available: http://doi.acm.org/10.1145/229542.229546

[16] S. Hunt and D. Sands, "Just Forget It: The Semantics and Enforcement of Information Erasure," in *ESOP'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 239–253. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792878.1792903

[17] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015," in *CANS 2016*, ser. LNCS, vol. 10052. Springer, 2016, pp. 573–582.

[18] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO '99*, ser. LNCS, vol. 1666. Springer, 1999, pp. 388–397.

[19] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: a verified implementation of ML," in *POPL '14*. ACM, 2014, pp. 179–192.

[20] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," in *POPL'06*. ACM, 2006, pp. 42–54. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111042

[21] ——, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[22] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, 2008.

[23] The Coq development team, *The Coq proof assistant reference manual*, 2017, version 8.7. [Online]. Available: http://coq.inria.fr

[24] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. LNCS, vol. 1384. Springer, 1998, pp. 151–166. [Online]. Available: https://doi.org/10.1007/BFb0054170

[25] S. Rideau and X. Leroy, "Validating register allocation and spilling," in *Compiler Construction, 19th International Conference, CC 2010*, ser. LNCS, vol. 6011. Springer, 2010, pp. 224–243.

[26] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009. [Online]. Available: https://doi.org/10.3233/JCS-2009-0352

[27] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *in PLDI 2013. ACM*, 2013.

[28] L. Simon, D. Chisnall, and R. J. Anderson, "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers," in *EuroS&P*. IEEE, 2018, pp. 1–15.

[29] ——, "What you get is what you C: controlling side effects in mainstream C compilers," in *EuroS&P*. IEEE, 2018, pp. 1–15.

[30] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. Kaashoek, "Undefined Behavior: What Happened to My Code?" in *APSYS '12*, 2012.

[31] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI 2011*. ACM, 2011, pp. 283–294.

[32] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead Store Elimination (Still) Considered Harmful," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USENIX Association, 2017, pp. 1025–1040. [Online]. Available: http://dl.acm.org/citation.cfm?id=3241189.3241269