Compiling Sandboxes: Formally Verified Software Fault Isolation

Frédéric Besson[1(✉)], Sandrine Blazy[1], Alexandre Dang[1], Thomas Jensen[1], and Pierre Wilke[2]

[1] Inria, Univ Rennes, CNRS, IRISA, Rennes, France
frederic.besson@inria.fr
[2] CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, Rennes, France

Abstract. Software Fault Isolation (SFI) is a security-enhancing program transformation for instrumenting an untrusted binary module so that it runs inside a dedicated isolated address space, called a sandbox. To ensure that the untrusted module cannot escape its sandbox, existing approaches such as Google's Native Client rely on a binary verifier to check that all memory accesses are within the sandbox. Instead of relying on *a posteriori* verification, we design, implement and prove correct a program instrumentation phase as part of the formally verified compiler CompCert that enforces a sandboxing security property *a priori*. This eliminates the need for a binary verifier and, instead, leverages the soundness proof of the compiler to prove the security of the sandboxing transformation. The technical contributions are a novel sandboxing transformation that has a well-defined C semantics and which supports arbitrary function pointers, and a formally verified C compiler that implements SFI. Experiments show that our formally verified technique is a competitive way of implementing SFI.

1 Introduction

Isolating programs with various levels of trustworthiness is a fundamental security concern, be it on a cloud computing platform running untrusted code provided by customers, or in a web browser running untrusted code coming from different origins. In these contexts, it is of the utmost importance to provide adequate isolation mechanisms so that a faulty or malicious computation cannot compromise the host or neighbouring computations.

There exists a number of mechanisms for enforcing isolation that intervene at various levels, from the hardware up to the operating system. Hypervisors [10], virtual machines [2] but also system processes [17] can ensure strong isolation properties, at the expense of costly context switches and limited flexibility in the interaction between components. Language-based techniques such as strong typing offer alternative techniques for ensuring memory safety, upon which access control policies and isolation can be implemented. This approach is implemented e.g. by the Java language for which it provides isolation guarantees, as proved by Leroy and Rouaix [21]. The isolation is fined-grained and very flexible but

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 499–524, 2019.
https://doi.org/10.1007/978-3-030-17184-1_18

the security mechanisms, e.g. stack inspection, may be hard to reason about [7]. In the web browser realm, JavaScript is dynamically typed and also ensures memory safety upon which access control can be implemented [29].

1.1 Software Fault Isolation

Software Fault Isolation (SFI) is an alternative for unsafe languages, e.g. C, where memory safety is not granted but needs to be enforced at runtime by program instrumentation. Pioneered by Wahbe *et al.* [35] and popularised by Google's Native Client [30,37,38], SFI is a program transformation which confines a software component to a memory sandbox. This is done by pre-fixing every memory access with a carefully designed code sequence which efficiently ensures that the memory access occurs within the sandbox. In practice, the sandbox is aligned and the sandbox addresses are thus of the form $0xYZ$ where Y is a fixed bit-pattern and Z is an arbitrary bit-pattern *i.e.*, $Z \in [0x0 \ldots 0, 0xF \ldots F]$. Hence, enforcing that memory accesses are within the sandbox range of addresses can be efficiently implemented by a *masking* operation which exploits the binary representation of pointers: it retains the lowest bits Z and sets the highest bits to the bit-pattern Y.

Traditionally, the SFI transformation is performed at the binary level and is followed by an *a posteriori* verification by a trusted SFI verifier [23,31,35]. Because the verifier can assume that the code has undergone the SFI transformation, it can be kept simple (almost syntactic), thereby reducing both verification time and the Trusted Computing Base (TCB). This approach to SFI can be viewed as a simple instance of Proof Carrying Code [25] where the compiler is untrusted and the binary verifier is either trusted or verified.

Traditional SFI is well suited for executing binary code from an untrusted origin that must, for an adequate user experience, start running as soon as possible. Google's Native Client [30,37] is a state-of-the-art SFI implementation which has been deployed in the Chrome web browser for isolating binary code in untrusted pages. ARMor [39] features the first fully verified SFI implementation where the TCB is reduced to the formal ARM semantics in the HOL proof-assistant [9]. RockSalt [24] is a formally verified implementation of an SFI verifier for the x86 architecture, demonstrating that an efficient binary verifier can be obtained from a machine-checked specification.

1.2 Software Fault Isolation Through Compilation

A downside of the traditional SFI approach is that it hinders most compiler optimisations because the optimised code no longer respects the simple properties that the SFI verifier is capable of checking. For example, the SFI verifier expects that every memory access is immediately preceded by a specific syntactic code pattern that implements the sandboxing operation. A semantically equivalent but syntactically different code sequence would be rejected. An alternative to the *a posteriori* binary verifier approach is Portable Software Fault Isolation (PSFI), proposed by Kroll *et al.* [16]. In this methodology, there is no verifier

to trust. Instead isolation is obtained by compilation with a machine-checked compiler, such as CompCert [18]. Portability comes from the fact that PSFI can reuse existing compiler back-ends and therefore target all the architectures supported by the compiler without additional effort.

PSFI is applicable in scenarios where the source code is available or the binary code is provided by a trusted third-party that controls the build process. For example, the original motivation for Proof Carrying Code [25] was to provide safe kernel extensions [26] as binary code to replace scripts written in an interpreted language. This falls within the scope of PSFI. Another PSFI scenario is when the binary code is produced in a controlled environment and/or by a trusted party. In this case, the primary goal is not to protect against an attacker trying to insert malicious code but to prevent honest parties from exposing a host platform to exploitable bugs. This is the case *e.g.* in the avionics industry, where software from different third-parties is integrated on the same host that needs to ensure strong isolation properties between tasks whose levels of criticality differ. In those cases, PSFI can deliver both security and a performance advantage. In Sect. 8, we provide experimental evidence that PSFI is competitive and sometimes outperforms SFI in terms of efficiency of the binary code.

1.3 Challenges in Formally Verified SFI

PSFI inserts the masking operations during compilation and does away with the *a posteriori* SFI verifier. The challenge is then to ensure that the security, enforced at an intermediate representation of the code, still holds for the running code. Indeed, compiler optimisation often breaks such security [33]. The insight of Kroll *et al.* is that a safety theorem of the compiled code (i.e., that its behaviour is well-defined) can be exploited to obtain a security theorem for that same compiled code, guaranteeing that it makes no memory accesses outside its sandbox. We explain this in more detail in Sect. 2.2.

One challenge we face with this approach is that it is far from evident that the sandboxing operations and hence the transformed program have well-defined behaviour. An unsafe language such as C admits undefined behaviours (e.g. bitwise operations on pointers), which means that it is possible for the observational behaviour of a program to differ depending on the level of optimisation. This is not a compiler bug: compilers only guarantee semantics preservation *if* the code to compile has a well-defined semantics [36]. Therefore, our SFI transformation must turn any program into a program with a well-defined semantics.

The seminal paper of Kroll *et al.* emphasises that the absence of undefined behaviour is a prerequisite but they do not provide a transformation that enforces this property. More precisely, their transformation may produce a program with undefined behaviours (*e.g.* because the input program had undefined behaviours). This fact was one of the motivation for the present work, and explains the need for a new PSFI technique. One difficulty is to remove undefined behaviours due to restrictions on pointer arithmetic. For example, bitwise operators on pointers have undefined C semantics, but traditional masking operations of SFI rely heavily on these operators. Another difficulty is to deal with

indirect function calls and ensure that, as prescribed by the C standard, they are resolved to valid function pointers. To tackle these problems, we propose an original sandboxing transformation which unlike previous proposals is compliant with the C standard [13] and therefore has well-defined behaviour.

1.4 Contributions

We have developed and proved correct CompCertSfi, the first full-fledged, fully verified implementation of SFI inside a C compiler. The SFI transformation is performed early in the compilation chain, thereby permitting the generated code to benefit from existing optimisations that are performed by the back-end. The technical contributions behind CompCertSfi can be summarised as follows.

- An original design and implementation of the SFI transformation based on well-defined pointer arithmetic and which supports function pointers. This novel design of the SFI transformation is necessary for the safety proof.
- A machine-checked proof of the **security** and **safety** of the SFI transformation. Our formal development is available online [1].
- A small, lightweight runtime system for managing the sandbox, built using a standard program loader and configured by compiler-generated information.
- Experimental evidence demonstrating that the portable SFI approach is competitive and sometimes even outperforms traditional SFI, in particular state-of-the-art implementations of (P)Native Client.

The rest of the paper is organised as follows. In Sect. 2, we present background information about the CompCert compiler (Sect. 2.1) and the PSFI approach (Sect. 2.2). Section 3 provides an overview of the layout of the sandbox and the masking operations implementing our SFI. In Sect. 4 we explain how to overcome the problem with undefined pointer arithmetic and define masking operations with a well-defined C semantics. Section 5 describes how control-flow integrity in the presence of function pointers can be achieved by a sligthly more flexible SFI policy which allows reads in well-defined areas outside the sandbox. Section 6 specifies the SFI policy in more detail, and describes the formal Coq proofs of safety and security. Section 7 presents the design of our runtime library and how it exploits compiler support. Experimental results are detailed in Sect. 8. Section 9 presents related work and Sect. 10 concludes.

2 Background

This section presents background information about the CompCert compiler [18] and the Portable Software Fault Isolation proposed by Kroll *et al.* [16].

2.1 CompCert

The CompCert compiler [18] is a machine-checked compiler programmed and proved correct using the Coq proof-assistant [22]. It compiles C programs down

$$
\begin{aligned}
\mathit{constant} \ni c &::= \mathit{i32} \mid \mathit{i64} \mid \mathit{f32} \mid \mathit{f64} \mid \&\mathit{gl} \mid \&\mathit{stk} \\
\mathit{chunk} \ni \kappa &::= \mathit{is}_8 \mid \mathit{iu}_8 \mid \mathit{is}_{16} \mid \mathit{iu}_{16} \mid \mathit{i}_{32} \mid \mathit{i}_{64} \mid \mathit{f}_{32} \mid \mathit{f}_{64} \\
\mathit{expr} \ni e &::= x \mid c \mid \triangleright e \mid e_1 \square e_2 \mid [e]_\kappa \\
\mathit{stmt} \ni s &::= \mathbf{skip} \mid x := e \mid [e_1]_\kappa := e_2 \mid \mathbf{return}\ e \mid x := e(e_1 \ldots, e_n)_\sigma \\
&\quad \mid \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid s_1; s_2 \mid \mathbf{loop}\ s \mid \{s\} \mid \mathbf{exit}\ n \mid \mathbf{goto}\ \mathit{lb}
\end{aligned}
$$

Fig. 1. CMINOR syntax

to assembly code through a succession of compiler passes which are shown to be semantics preserving. COMPCERT features an architecture independent front-end. The back-end supports four main architectures: x86, ARM, PowerPC and RiscV. To target all the back-ends without additional effort, our secure transformation is performed in the compiler front-end, at the level of the CMINOR language that is the last architecture-independent language of the COMPCERT compiler chain. Our transformation can obviously be applied on C programs by first compiling them into CMINOR, and then applying the transformation itself.

The CMINOR language is a minimal imperative language with explicit stack allocation of certain local variables [19]. Its syntax is given in Fig. 1. Constants range over 32-bit and 64-bit integers but also IEEE floating-point numbers. It is possible to get the address of a global variable *gl* or the address of the stack allocated local variables (i.e., *stk* denotes the address of the current stack frame). In COMPCERT parlance, a memory chunk κ specifies how many bytes need to be read (resp. written) from (resp. to) memory and whether the result should be interpreted as a signed or unsigned quantity. For instance, the memory chunk is_{16} denotes a 16-bit signed integer and f_{64} denotes a 64-bit floating-point number. In CMINOR, memory accesses, written $[e]_\kappa$, are annotated with the relevant memory chunk κ. Expressions are built from pseudo-registers, constants, unary ($\triangleright$) and binary ($\square$) operators. COMPCERT features the relevant unary and binary operators needed to encode the semantics of C. Expressions are side-effect free but may contain memory reads.

Instructions are fairly standard. Similarly to a memory read, a memory store $[e_1]_\kappa = e_2$ is annotated by a memory chunk κ. In CMINOR, a function call such as $e(e_1 \ldots, e_n)_\sigma$ represents an indirect function call through a function pointer denoted by the expression e, σ is the signature of the function and $e_1 \ldots, e_n$ are the arguments. A direct call is a special case where the expression e is a constant (function) pointer. CMINOR is a structured language and features a conditional, a block construct $\{s\}$ and an infinite loop **loop** s. Exiting the n^{th} enclosing loop or block can be done using an **exit** n instruction. CMINOR is structured but **goto**s towards a symbolic label *lb* are also possible. Returning from a function is done by a return instruction. CMINOR is equipped with a small-step operational semantics. The intra-procedural and inter-procedural control flows are modelled using an explicit continuation which therefore contains a call stack.

CompCert Soundness Theorem. Each compiler pass is proved to be semantics preserving using a simulation argument. Theorem 1 states semantics preservation.

Theorem 1 (Semantics Preservation). *If the compilation of program p succeeds and generates a target program tp, then for any behaviour beh of program tp there exists a behaviour of p, beh′, such that beh* improves *beh′.*

In this statement, a behaviour is a trace of observable events that are typically generated when performing external function calls. CompCert classifies behaviours depending on whether the program terminates normally, diverges or goes wrong. A *goes wrong* behaviour corresponds to a situation where the program semantics gets stuck (i.e., has an undefined behaviour). In this situation, the compiler has the liberty to generate a program with an *improved* behaviour i.e., the semantics of the transformed program may be more defined (i.e., it may not get stuck at all or may get stuck later on).

The consequence is that Theorem 1 is not sufficient to preserve a safety property because the target program *tp* may have behaviours that are not accounted for in the program *p* and could therefore violate the property. Corollary 1 states that in the absence of going-wrong behaviour, the behaviours of the target program are a subset of the behaviours of the source program.

Corollary 1 (Safety preservation). *Let p be a program and tp be a target program. Consider that none of the behaviours of p is a going-wrong behaviour. If the compilation of p succeeds and generates a target program tp, then any behaviour of program tp is a behaviour of p.*

As a consequence, any (safety) property of the behaviours of p is preserved by the target program *tp*. In Sect. 2.2, we show how the PSFI approach leverages Corollary 1 to transfer an isolation property obtained at the Cminor level to the assembly code.

Going-wrong behaviours in CompCert. As safety is an essential property of our PSFI transformation, we give below a detailed account of the going-wrong behaviours of the CompCert languages with a focus on Cminor.

Undefined evaluation of expressions. CompCert's runtime values are dynamically typed and defined below:

$$\mathit{values} \ni v ::= \mathbf{undef} \mid \mathbf{int}(i_{32}) \mid \mathbf{long}(i_{64}) \mid \mathbf{single}(f_{32}) \mid \mathbf{float}(f_{64}) \mid \mathbf{ptr}(b, o)$$

Values are built from numeric values (32-bit and 64-bit integers and floating point numbers), the **undef** value representing an indeterminate value, and pointer values made of a pair (b, o) where b is a memory block identifier and o is an offset which, depending on the architecture, is either a 32-bit or a 64-bit integer.

For Cminor, like all languages of CompCert, the unary ($\triangleright$) and binary ($\square$) operators are not total. They may directly produce going-wrong behaviours *e.g.* in case of division by $\mathbf{int}(0)$. They may also return **undef** if (i) the arguments are not in the right range *e.g.* the left-shift $\mathbf{int}(i) << \mathbf{int}(32)$; or (ii) the arguments are not well-typed *e.g.* $\mathbf{int}(i) +_{int} \mathbf{float}(f)$. Pointer arithmetic is strictly conforming to the C standard [13] and any pointer operation that is implementation-defined according to the standard returns **undef**.

$$
\begin{array}{ll}
\mathbf{ptr}(b,o) \pm \mathbf{long}(l) & = \mathbf{ptr}(b, o \pm l) \\
\mathbf{ptr}(b,o) - \mathbf{ptr}(b,o') & = \mathbf{long}(o - o') \\
\mathbf{ptr}(b,o)\,!{=}\mathbf{long}(0) & = \mathbf{tt} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) == \mathbf{long}(0) & = \mathbf{ff} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) \star \mathbf{ptr}(b,o') & = o \star o' \quad \text{if } W(b,o) \wedge W(b,o') \\
\mathbf{ptr}(b,o) == \mathbf{ptr}(b',o') & = \mathbf{ff} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\mathbf{ptr}(b,o)\,!{=}\mathbf{ptr}(b',o') & = \mathbf{tt} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\multicolumn{2}{c}{\text{where } \star \in \{<, \leq, ==, \geq, >, \,!{=}\}}
\end{array}
$$

Fig. 2. Pointer arithmetic in CompCert

The precise semantics of pointer operations is given in Fig. 2. For simplicity, we provide the semantics for a 64-bit architecture. Pointer operations are often only defined provided that the pointers are valid, written V, or weakly valid, written W. This validity condition requires that the offset o of a pointer $\mathbf{ptr}(b,o)$ is strictly within the bounds of the block b. The weakly valid condition refers to a pointer whose offset is either valid or one-past-the-end of the block b. Any pointer arithmetic operation that is not listed in Fig. 2 returns **undef**. This is in particular the case for bitwise operations which are typically used for the masking operation needed to implement SFI.

The indeterminate value **undef** is not *per se* a going-wrong behaviour. Yet, branching over a test evaluating to **undef**, performing a memory access over an **undef** address and returning **undef** from the `main` function are going-wrong behaviours.

Memory accesses are ruled by a unified memory model [20] that is used throughout the whole compiler. The memory is made of a collection of separated blocks. For a given block, each offset o below the block size is given a permission $p \in \{\mathbf{r}, \mathbf{w}, \dots\}$ and contains a memory value

$$mval \ni mv ::= \mathbf{undef} \mid \mathbf{byte}(b) \mid [\mathbf{ptr}(b,o)]_n$$

where b is a concrete byte value and $[\mathbf{ptr}(b,o)]_n$ represents the n^{th} byte of the pointer $\mathbf{ptr}(b,o)$ for $n \in \{1 \dots 8\}$. A memory write $storev(\kappa, m, a, v)$ is only defined if the address a is a pointer $\mathbf{ptr}(b,o)$ to an existing block b such that the memory locations $(b,o), \dots, (b, o+ \mid \kappa \mid -1)$ have the permission $\mathbf{w}$ and the offset o satisfies the alignment constraint of κ. A memory read $loadv(\kappa, m, a)$ is only defined under similar conditions with the additional restriction that not reading all the consecutive fragments of a pointer returns **undef**.

Control-flow transfers may go-wrong if the target of the control-flow transfer is not well-defined. Hence, a **goto** lb instruction goes wrong if, in the current function, there is no statement labelled by lb; and an **exit** n instruction goes wrong if there are less than n enclosing blocks around the statement containing the exit instruction. A conditional **if** e **then** s_1 **else** s_2 goes wrong if the expression e does not evaluate to $\mathbf{int}(i)$ for some i. Also, the execution goes wrong if the

last statement of a function is not a **return** instruction. Last but not least, a function call $x := e(e_1 \ldots, e_n)_\sigma$ goes wrong if the expression e does not evaluate to a pointer $\mathbf{ptr}(b, 0)$ where b is a function pointer with signature σ.

We show in Sect. 4 how our transformation ensures that pointer arithmetic and memory accesses are always well-defined. Section 5 shows how we make sure indirect calls are always correctly resolved. Section 6 shows that, together with other statically checkable verifications, our PSFI transformation rules out all possible going-wrong behaviours.

2.2 Portable Software Fault Isolation

Kroll, Stewart and Appel have pioneered the concept of Portable Software Fault Isolation (PSFI) [16] whereby SFI is enforced by a pass of the compiler front-end that is architecture independent. The main expected advantage is that isolation is implemented, once and for all, for any target architecture. Moreover, the generated code is optimised by the back-end passes of the compiler. Compared to traditional SFI, there is no architecture-specific binary verifier but instead the compiler enters the TCB. The key insight of Kroll *et al.* is to leverage a formally verified compiler, namely CompCert, to transfer a security proof of isolation obtained at the Cminor level through the compiler back-end, with minimal proof effort. In the following, we recall the only basic properties that a Cminor SFI transformation needs to satisfy so that isolation holds at assembly level.

In CompCert's terms, the sandbox is identified by a dedicated memory block *sb*. A Cminor program is secure (Property 1) under the condition that all its memory accesses are performed within the sandbox.

Property 1 (Program security). A Cminor program p is secure if all its memory accesses are within the sandbox block *sb*.

After compilation, the assembly code is secure if its observable behaviours are the same as the observable behaviours of the Cminor program. In order to apply CompCert's semantics preservation theorem (more precisely Corollary 1), it remains to ensure that the Cminor program has a well-defined semantics (Property 2).

Property 2 (Program safety). A Cminor program p is safe if all its behaviours are well-defined, i.e., not wrong.

Kroll *et al.* state Property 1 by means of an instrumented Cminor semantics which gets stuck in case of memory accesses outside the sandbox. They prove formally that the additional semantic safeguards are never triggered for a transformed program.

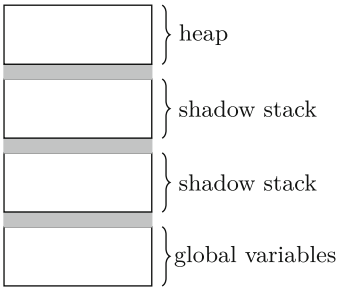
Kroll *et al.* also sketch some necessary steps to prove the Property 2 of safety but do not propose a formal proof. This leaves open a number of challenging issues such as whether it is feasible to define a masking operation that has a defined Cminor semantics and how to deal with indirect function calls through function pointers, More generally, the work leaves open whether a formal proof

of Property 2 on safety is possible given the restrictions of CompCert's semantics (notably pointer arithmetic) and without relying on axioms asserting properties of an external masking primitive. One of the central contributions of this work is to provide a positive answer to this question and propose solutions to these issues where neither the sandboxing of memory accesses nor the sandboxing of function pointers is part of a TCB. The transformation that circumvents the limitations imposed by pointer arithmetic is original and, we surmise, is a necessary component to transfer security down to assembly. For a precise comparison with Kroll *et al.* see Sect. 9).

3 A Thread-Aware Sandbox

The memory address space of a C program is partitioned into a runtime stack of frames, a heap and a dedicated space for global variables. The address space of a sandboxed program is re-organised to fit into a single global variable, *sb*, where the global variables, the heap and the stack frames are relocated. Figure 3a depicts the memory layout of the program after our SFI transformation. Each global variable is relocated and allocated in the sandbox at a given offset, and each global memory access of the program is translated into a memory access in the sandbox. For managing the heap it suffices to use a sandbox-aware `malloc` implementation that allocates memory inside the sandbox.

To prevent buffer overflows, a standard approach consists in introducing a so-called *shadow stack* that is used to store the function stack frames. Our implementation supports multi-threaded applications and therefore there are as many shadow stacks as there are threads. Upon thread creation, we allocate a novel shadow stack in the sandbox. The shadow-stack pointer is passed as an additional argument to each function call. This is efficient when arguments are passed by register, with the only drawback of reserving an additional register. Frames are allocated by incrementing the shadow-stack pointer at function entry. All accesses to the original stack are then translated into accesses to the sandbox shadow stack. The following Example 1 and the code snippet in Fig. 3 illustrate the essence of the transformation.



(a) Layout of memory

```
g = {long(5)};

long foo(){
  stk[8];
  bar(g, &stk);
  return([&stk]);
}
```

(b) Original Cminor

```
sb[2^k]= {long(5);...};

long foo(sp){
  sp1=sp + 8 ;
  bar(sp1,[&sb],sp);
  return([sp]);
}
```

(c) Sandboxed Cminor

Fig. 3. Sandbox transformation

Example 1. The Cminor program of Fig. 3b declares a global variable `g` initialised to the 64-bit integer 5. The function `foo` allocates a stack frame of 8 bytes that will be used to store a 64-bit local variable. By convention, the current stack frame is called `stk`. The function `foo` calls the function `bar` with as arguments the value of `g` and the address of the local variable `stk`; and returns the value, presumably updated by `bar`, of the local variable.

Syntactically, the program of Fig. 3c only performs memory accesses on the global sandbox `sb` variable. The size of `sb` variable is 2^k for some predefined k. At thread creation, a shadow stack is allocated by our sandbox-aware `malloc` in the sandbox after the statically allocated global variables. For our program, the unique global variable `g` is stored at offset 0 and spans over 8 bytes. Therefore, the initial value of the shadow-stack pointer `sp` is 8. After the transformation, the function `foo` reserves the space for the local variable `stk` by incrementing the pseudo-register `sp`. The function `bar` is called with the incremented shadow-stack pointer `sp1`, the value stored at offset 0 in the sandbox (i.e., the value of the global variable `g`) and the address of the local variable `stk` which is given by the value of the stack pointer `sp`. At function exit, the value of the local variable `stk` is returned by dereferencing the shadow-stack pointer `sp`.

Our SFI transformation enforces the isolation security policy stipulating that all memory accesses are performed within the sandbox *sb*—at the Cminor level. However, this holds because the semantics gets stuck (i.e., the semantics *goes wrong*) whenever the program performs an access outside the bounds of the sandbox. As explained earlier, the compiler is free to translate this into an insecure program that would escape the sandbox at runtime. To get a formal security guarantee, it is necessary to transform further the Cminor program to rule out any behaviour that *goes wrong* i.e., ensure Property 2. Given the numerous undefined behaviours of the C language, ruling out any *going-wrong* behaviour may seem a daunting task. In general, this requires to ensure both memory safety and control-flow integrity. The following two sections describe how we can exploit the SFI transformation and the knowledge that all memory accesses are inside the sandbox to ensure both memory safety and control-flow integrity.

4 Memory-Safe Masking

For SFI, memory safety is obtained by making sure that every memory access is performed inside the sandbox. Starting from an analysis of the standard SFI solution, we present our own design which satisfies the additional requirements of being compliant with the semantic restrictions of CompCert and with a strict interpretation of the C standard.

4.1 Standard SFI Masking of Addresses

Standard SFI transformations ensure memory safety by masking memory accesses. The gist of it is to allocate a sandbox *sb* of size 2^k at a 2^k aligned memory address, say $\&sb = tag \times 2^k$. Under those constraints, enforcing that an address A is within the bounds of the sandbox can essentially be done by replacing the high-address bits by those of *tag*. Using bitwise operations, this can be done by the expression $(A\&(2^k-1))|tag \times 2^k$, where & is the bitwise *and* and | is the bitwise *or*. More visually, this can be written $(A\&\underbrace{1\cdots1}_{k})|tag\,\underbrace{0\cdots0}_{k}$.

At binary level, this masking transformation is defined and the cost is modest: two bitwise operations. However, this masking operation has no well-defined C semantics. This is also the case for the semantics of CompCert and in particular for the Cminor language. The reason is twofold: bitwise operations over pointer values return **undef** and concrete addresses (e.g. $tag \times 2^k$) are not pointers for CompCert where they are represented by a block and an offset (see Fig. 2).

4.2 Specialised Masking for 32-Bit Sandboxes

For 32-bit sandboxes, there exists a variant of the sandboxing primitive which has the advantages (1) that the sandbox address does not need to be aligned; (2) that the cost of masking may be reduced to a single instruction. In its simplest form, the masking primitive is defined by

$$\&sb + (A - \&sb)_{64\to32\to64}$$

where $\&sb$ is the symbolic address of the sandbox. The subtraction of $\&sb$ extracts the offset of the pointer and the double (unsigned) cast $64 \to 32 \to 64$ has the effect of truncating the offset to a 32-bit quantity that is therefore within the bounds of a 32-bit sandbox. At first sight, this masking is less efficient than the standard masking but it is efficient for typical address computations which require both displacement and scaling (e.g. $A = t + k + k' * i_{32\to64}$ where t is a 64-bit address, k and k' are constants and i is a 32-bit integer). Assuming that each cast or arithmetic operation is mapped to a single instruction[1], the masked address A can be computed using 8 instructions: 4 instructions for computing the address A and 4 more for the sandboxing primitive. Using simple properties of modular arithmetic, it is possible to distribute the $64 \to 32$ cast over addition and multiplication to obtain the following equivalent formulation of the sandboxed address:

$$\&sb + A'_{32\to64} \quad with \quad A' = t_{64\to32} + c_1 + c_2 * i$$

where c_1 and c_2 are compile-time constants: $c_1 = (k - \&sb)_{64\to32}$ and $c_2 = k'_{64\to32}$. Using this formulation, the address A' still requires 4 instructions but the cost of the sandboxing is reduced to 2 instructions making it on par with the standard sandboxing. On x86, 32-bit registers are just zero-extended 64-bit registers. Therefore, the cast $A'_{32\to64}$ is actually redundant and the overhead induced by the sandboxing is reduced to a single instruction. Our experiments (see Sect. 8.2) validate the practical advantage of this encoding.

Still, as for the standard sandboxing, this sanboxing primitive has no semantics in CompCert due to the limitations of pointer arithmetic. As a consequence, the solution of Kroll *et al.* [16] does not give actual code for the masking primitive, but rather axiomatise its behaviour as an external function. This prevents optimisations such as common subexpression elimination or function inlining from happening and induces the cost of a function call for each memory access.

4.3 Towards Well-Defined Pointer Arithmetic

To illustrate the limitations of pointer arithmetic, we examine the semantic behaviour of the standard sandboxing primitive (the specialised sandboxing primitive has similar

[1] Some architecture have rich addressing modes allowing for more compact encodings.

issues). The standard sandboxing primitive can be written $(A\&(2^k-1)) \mid \&sb$ where $\&sb$ is the address of the sandbox variable. If sb is allocated at runtime at address $tag \times 2^k$ for some tag, this formulation is equivalent at binary level. Again, this heavily relies on pointer arithmetic that is undefined and on information about where the sandbox is linked at runtime.

Consider the alternative formulation $(A\&(2^k-1)) + \&sb$ where the bitwise $\mid$ is replaced by a $+$. This formulation has the advantage that incrementing a pointer, here sb, is well-defined (see Fig. 2). As on modern hardware, both addition and bitwise operations take a single cycle, the difference in efficiency should be negligible. Moreover, at least for x86, the addition can be compiled into the addressing mode.

Still, this does not solve our issue. To understand this, suppose that A is a pointer. In this case, the bitwise $\&$, whose purpose is to extract the pointer offset, is still undefined. Therefore, the whole expression $(A\&(2^k-1)) + \&sb$ is undefined. Because dereferencing an undefined expression is a *going-wrong* behaviour, the compiled program may have an arbitrary runtime behaviour and escape the sandbox. A prerequisite for our masking primitive is therefore to ensure that the evaluation is defined i.e., different from **undef**. As all the semantic operators of CompCert are strict in **undef** (if any argument is **undef**, so is the result), a necessary condition is that A is not **undef**. As A can be obtained from any expression, a challenge is to ensure that every expression evaluates to a defined value. A particular difficulty is that the many undefined pointer operations (see Fig. 2) cannot be detected by runtime checks.

4.4 Arithmetisation of the Heap

To tackle this challenge and ensure that every computation is defined, we propose an original and radical approach which ensures syntactically that pointers are neither stored in memory nor in local variables. As a result, the program is only manipulating integer values and memory addresses are only constructed by the sandboxing primitives. This approach implies, as a side-effect, that our previously undefined masking primitives are defined. Let asb be the runtime address of the symbolic address $\&sb$ of the sandbox. The masking of an address A can be written

$$A' + \&sb$$

where A' is either defined by $A' = A\&(2^k-1)$ or $A' = (A - asb)_{64\to 32\to 64}$. As A is necessarily an integer, A' is necessarily a defined integer and therefore $A' + \&sb$ returns a defined pointer $\mathbf{ptr}(sb, o)$ that is necessarily inside the sandbox.

An additional subtlety is that memory accesses are indexed by a memory chunk κ which mandates an alignment constraint (e.g. the chunk i_{64} mandates an 8-byte aligned address). As a result, the masking primitive is parameterised by the chunk κ and the masking primitive for i_{64} is $A'\&msk_{i_{64}} + \&sb$ where $msk_{i_{64}} = (2^{k-3}-1) \times 2^3$.

Only computing over numeric values is facilitated by the fact that the sandboxed program is only manipulating pointers relative to a single object, the sandbox. Therefore, a solution could be to only compute with pointer offsets. This is not totally satisfactory because the null pointer (i.e., 0) would be undistinguishable from the base pointer $\mathbf{ptr}(sb, 0)$. Instead, we use the integer asb that is the integer runtime address of the sandbox (i.e., we have $asb = \&sb$) and perform the following transformation t over program expressions.

$$\begin{array}{ll} t(\&sb) & = asb \\ t(c) & = c \text{ for } c \in \{i32, i64, f32, f64\} \\ t(\triangleright e) & = \blacktriangleright\, t(e) \\ t(e_1 \square e_2) & = t(e_1) \blacksquare t(e_2) \\ t([e]_\kappa) & = [msk_\kappa(t(e))] \end{array}$$

The operators ▶ and ■ ensure that, if the expressions are well-typed, they never return the **undef** value. Typical examples include division, modulus, and bitwise shifts. We transform expressions so that they evaluate to an arbitrary value when their original semantics is undefined. For example, we transform the left-shift operations on 32-bit integers so that the resulting expression always has a shift amount less than 32:

$$\texttt{a≪b} \quad \rightsquigarrow \quad \texttt{a ≪ (b \& 31).}$$

Similarly, we transform divisions and modulus in the following way, to rule out the undefined cases of division by zero and signed division of `MIN_SIGNED` by `-1`:

$$\texttt{a/b} \rightsquigarrow \texttt{(a+(a==MIN\_SIGNED \& b==-1))/(b+(b==0)).}$$

We can prove that the resulting division expression is always defined. Most of the other expressions are always defined and do not need further transformations.

5 Enforcement of Control-Flow Integrity

Correct sandboxing of code requires some degree of control-flow integrity. Existing SFI implementations enforce a weak form of control-flow integrity which only ensures that jumps are aligned and within a sandbox of code. This is achieved by inserting a masking operation before indirect jumps, that will mask the target address to ensure that the jump is within the sandbox. Additional padding with no-ops is inserted to ensure that all the instructions are indeed aligned [30,37,38]. We enforce a stronger, more traditional, form of control-flow integrity where any control-flow transfer has a well-defined Cminor semantics.

5.1 Relaxation of the Cminor SFI Property

Intraprocedural control-flow integrity is ensured by simple syntactic checks. For instance, they ensure that a **goto** *lb* has a corresponding label *lb* and that an **exit** *n* has at least *n* enclosing blocks. The semantics of Cminor prescribes that function calls and returns necessarily match. For this to still hold at the assembly level where the return address is explicitly stored in the stack frame, it is sufficient to prove that the Cminor program has no *going-wrong* behaviour. To ensure control-flow integrity, the only remaining issue is due to indirect calls through function pointers. Our control-flow integrity counter-measure implements software trampolines and ensures that an indirect call with signature σ can only be resolved by a function pointer towards a function with signature σ.

For this purpose, the existing Cminor SFI security policy i.e., Property 1, which rules out any memory access outside the sandbox is too restrictive. As we shall see, the implementation of trampolines necessitates controlled memory reads, outside the sandbox, within compiler-generated variables. To accommodate for this extension, we propose a slightly relaxed SFI security property which, in addition to memory accesses inside the sandbox, authorises other memory reads in read-only regions.

Property 3. A CMINOR program is secure if all its memory accesses are within either the sandbox block *sb* or some read-only memory.

This relaxed property still ensures the integrity of the runtime because all memory writes are confined to the sandbox. Note that Property 3 and Property 1 are equivalent if the trusted runtime library has no read-only memory. This can be achieved at modest cost by modifying slightly the source code and remove the C type qualifier `const` which instructs the compiler that the memory is read-only.

5.2 Control-Flow Integrity of Indirect Calls

In Sect. 4, we have eluded the presence of function pointers. They actually perfectly fit our strategy of encoding pointers by integers. In this case, each function pointer is encoded as an index and the trampoline code translates the index into a valid function pointer.

Consider a function f of signature σ and suppose that the function pointer $\&f$ is compiled into the index i. The reverse mapping from indexes to function pointers is obtained from a compiler-generated array variable A_σ such that $A_\sigma[i] = \&f$. The array variable A_σ is made of all the function pointers with signature σ. The array variable is also padded with a default function pointer such that its length is a power of two. At the call site, the instruction $e(e_1 \ldots, e_n)_\sigma$ is transformed into $[te\&msk_\sigma + \&A_\sigma](te_1, \ldots, te_n)_\sigma$ where $te, te_1 \ldots, te_n$ are transformed expressions such that all memory accesses are masked and msk_σ is the binary mask ensuring that the index te is within the bounds of the variable A_σ. In our actual implementation, we optimise direct calls and in this case bypass the trampoline. Therefore, when the expression e is a constant pointer $\&f$ to an existing function with signature σ, we generate directly $(\&f)(te_1 \ldots, te_n)$. As a result, only C code using indirect calls goes through the trampoline code.

Though our implementation only exploits the relaxation of Property 3 for the sake of trampolines, a more aggressive implementation could sometimes avoid to relocate read-only memory inside the sandbox. This could have a positive impact on optimisations which exploit the immutability of read-only memory.

6 Safety and Security Proofs

We next give an overview of our fully verified Coq proof of security and safety.

6.1 Security Proof

Property 3 is an informal formulation of our security property that is formally stated as a CMINOR instrumented semantics. This semantics mimics the CMINOR semantics with the exception that memory accesses are restricted: a memory read is either performed within the sandbox or in a read-only memory region; a memory write is necessarily performed within the sandbox.

The goal of the security proof is to show that all the memory accesses abide by the restrictions of the instrumented semantics. This is stated by Theorem 2 which establishes that for a transformed program *tp*, no behaviour of the standard CMINOR semantics gets stuck for the instrumented CMINOR semantics.

Theorem 2 (Security). *For any transformed program tp, every behaviour of tp in the standard semantics of* Cminor *is also a behaviour of tp in the instrumented semantics.*

The proof is based on the standard technique of forward simulation that is used in CompCert to ensure the preservation of semantics by compiler passes. Here, the forward simulation has the distinctive feature of relating the same (transformed) program equipped with a standard and an instrumented semantics. Since the only difference between the two semantics is that memory accesses must be secure, the crux of the proof lies in the correctness of the masking primitive, as stated in the following lemma.

Lemma 1. *For any masked expression e, if e evaluates to some pointer* $\mathbf{ptr}(b, o)$*, then b is the block of the sandbox i.e., sb.*

The proof relies on the definition of the masking primitive: a masked expression e is of the form $e' + \&sb$. Since $\&sb$ evaluates to the pointer $\mathbf{ptr}(sb, 0)$, then if the whole expression evaluates to a pointer $\mathbf{ptr}(b, o)$, necessarily $b = sb$.

6.2 Safety Proof

In order to benefit from CompCert's semantic preservation theorem and transport our security proof to the compiled assembly program, we must also prove that the sandboxed program is safe, i.e., it never gets *stuck*. We address all the going-wrong behaviours that we enumerated in Sect. 2.1. The well-formedness properties of a program (calling only defined functions, accessing only defined variables, jumping only to defined labels, exiting from no more blocks than currently enclosed in) are checked statically and make the transformation fail if they are violated. Next, the memory accesses require the addresses to be valid and adequately aligned: our masking operation ensures that this is always the case. Then, the evaluation of expressions must always be defined: this has mostly been dealt with the arithmetisation of the memory (Sect. 4.4). Finally, function calls should always be performed with the appropriate number of well-typed arguments. This is easy to check statically for direct function calls, but requires trampolines (as described in Sect. 5.2) for indirect function calls. The following sandbox invariant encapsulates all these conditions.

Definition 1 (Sandbox Invariant). *A state S of program P satisfies the sandbox invariant if the following conditions are satisfied:*

1. *indirect control-flow transfers are well-defined in P (e.g.* `goto` *instructions in the functions of P only jump to defined labels);*
2. *every function of P ends with an explicit return;*
3. *every function of P is well-typed;*
4. *every function of P starts by explicitly initialising its local variables;*
5. *the global array* A_σ *for signature* σ *contains function pointers to functions of signature* σ*;*
6. *the environment for local variables and the memory in S only contain properly initialised, numerical values.*

Properties 1, 2, 3 are ensured by a set of syntactic checks over the bodies of all the functions of the program. Property 4 is enforced by our function transformation which inserts assignments that explicitly initialise all declared local variables. Property 5 is ensured by construction of the arrays for function pointers. All these properties can be established solely on the program body and do not change during the execution of the program. By contrast, Property 6 cannot be checked statically and depends on the state of the program at each point.

Safe Evaluation of Expressions. A necessary condition for the safe evaluation of expressions is that the program is well typed. CompCert does not generate these type guarantees so we have integrated a verified (simple) type-inference algorithm for Cminor programs. Type-checking alone is not sufficient to rule out undefined behaviours of C operators, but together with the transformations explained in Sect. 4.4, we prove the following lemma about the evaluation of transformed expressions.

Lemma 2 (Safe evaluation of expressions). *In a memory state and a well-typed environment for local variables containing only defined numerical values, the transformation of any well-typed expression e evaluates to a defined numerical value.*

Lemma 2 follows directly from the properties of our expression transformation.

Safety of Calls through Trampolines. As mentioned in Sect. 5, we implement software trampolines to secure function calls through function pointers. To ensure the safety of indirect function calls, we maintain a map *smap* from function signatures to the corresponding array identifier and the length of this array. The proof of safety relies on the fact that for every function f of signature σ present in a program, we have $smap(\sigma) = (A_\sigma, l_\sigma)$ such that all offsets lower than l_σ in A_σ contain a pointer to a function of signature σ. The safety proof of indirect calls itself is not hard, but we need to set up this signature map and establish invariants relating it to the global environment of the program.

Safety Theorem. Considering the invariants defined in Definition 1, we prove Lemma 3 which is our main technical result.

Lemma 3 (Safety). *For any* Cminor *program state S that satisfies the invariants, either S is a final state or there exists a sequence of steps from S to some S' such that S' also satisfies the invariants.*

A subtlety of the proof is that at function entry, the local variables carry the value **undef** and therefore the sandbox invariant only holds after they have been initialised by a sequence of assignments (see Property 4 of Definition 1).

Using Lemma 3, we can show Property 2, in the form of Theorem 3.

Theorem 3 (Safety of the transformation). *All behaviours of the transformed program are well-defined, i.e., not wrong.*

Proof. A going-wrong behaviour occurs precisely when a state is reached, from which no further step can be taken, though it is not a final state. Lemma 3, together with a proof that the initial state of the transformed prorgam satisfies the invariants, tells us that no such reachable state exists, concluding the proof. □

As a result, we benefit from CompCert's semantic preservation theorem and can transport the security proof down to the assembly program.

Theorem 4 (Security of the compiled program). *Let p be a transformed* Cminor *program. If p compiles into the assembly program tp, then tp is secure.*

The proof uses Corollary 1 and Theorem 2 to conclude that the behaviours of tp are the same as those of p, and hence secure.

7 SFI Runtime and Library

Our modified CompCert compiler, CompCertSfi, takes as input a C program unit in the form of a list of C files. Each C file is first compiled down to the Cminor language using the existing passes of the CompCert compiler. Then, all the Cminor programs are syntactically linked [14] together to form the program unit to be isolated inside the sandbox. CompCertSfi comes with a lightweight runtime and a generic support for interfacing with a trusted library (e.g. a libC). An originality of our approach is that the runtime is using a standard program loader. Moreover, the runtime gets some of its configuration through compiler-generated variables.

7.1 Loading the SFI Application

The sandboxed code is linked with our runtime library by a linker script which specifies where to load at runtime the *sb* variable, viewed as the data segment. The compiler also emits a sandbox configuration map which contains the symbolic address of the sandbox, its numeric value at runtime, the total size of the sandbox and the range of addresses reserved for global variables.

Our runtime code is executed before starting the sandboxed `main` function. It first checks that the sandbox is properly linked according to the sandbox configuration map, sets the shadow-stack pointer and initialises the sandbox heap using our sandbox-aware implementation of `malloc` based on `ptmalloc3`[2].

By construction, our runtime stack is free of buffer overruns. Yet, if the recursion is too deep, the stack may overflow. Therefore, the runtime inserts an unmapped page guard at the bottom of the stack and intercepts the segmentation fault. This protection suffices provided that the size of each function stack frame does not exceed a page; which can be checked at compile-time. Eventually, after copying its arguments inside the sandbox, the runtime calls the `main` function of the sandboxed application.

7.2 Monitoring Calls to the Runtime Library

The runtime library is trusted and therefore part of the TCB. To ensure isolation, each call towards the runtime library is monitored to check the validity of the arguments. For this purpose, a call to a library function, say `foo`, is renamed in the object file into a call to a function `sb_foo` which sanitises its arguments before really calling the function `foo`. The verifications are library specific but usually straightforward to implement. For `stdio`, the `FILE` structures are allocated by the runtime outside of the sandbox. Hence, the returned `FILE*` cannot be dereferenced to corrupt the `FILE` structure. To prevent the sandboxed program to forge `FILE*` pointers, the runtime maintains at all time the set of valid `FILE*`. For variadic functions *e.g.*, `printf`, we statically compile the format into a sequence of safe primitive calls. (We reject programs using formats computed at runtime). For functions in `string`, we check beforehand that the range of memory accesses is within the range of the sandbox. We also allow callbacks and therefore a runtime function may take a function pointer as argument. To ensure that the function is valid, the runtime is using the trampoline programming pattern presented in Sect. 5.2.

[2] http://www.malloc.de/malloc/ptmalloc3-current.tar.gz.

7.3 Communication via Global Variables

Programs may not only communicate *via* function calls but also directly *via* global variables. For the libC, this includes e.g. `stdout` or `errno`. To ensure isolation, CompCertSfi relocates those variables inside the sandbox but also generates a global variable map which is an array variable of the form

$$\{\&n_1, o_1, \ldots, \&n_i, o_i, \ldots, \&n_m, o_m\}$$

where $\&n_i$ is the symbolic address of a global variable and o_i is its offset in the sandbox. Using this information, the runtime has the ability to synchronise the values of the variables inside and outside the sandbox. For example, at program startup, the value of `stdout` (a `stream` pointer) is copied inside the sandbox at the relevant offset. This allows the sandboxed program to call `stdio` functions but protects the integrity of the stream. For `errno`, it is the responsibility of each runtime library call to synchronise the value of `errno` in the sandbox.

8 Experiments

We have evaluated our PSFI approach over the CompCert benchmark suite and a port of Quake. All the experiments have been carried over a quad-core Intel 6600U laptop at 2.6 GHz with 16 GB of RAM running Linux Fedora 27. For Quake, we explain how to adapt the code to our runtime library and verify the absence of noticeable slowdown. For the other benchmarks, we make a more detailed performance evaluation and compare CompCertSfi with CompCert, gcc, clang but also the state-of-the-art (P)NaCl implementation of SFI. In our experiments, all the benchmarks are ordered by increasing running time. Moreover, for computing a runtime overhead, the running time is obtained by taking the harmonic mean of 3 consecutive runs.

8.1 Porting Quake

Quake engines come in various flavours and we use the tyr-quake[3] implementation linking with Xlib. The port requires the addition of several functions to our runtime library from Xlib and the libC. Most of them are not problematic and require no or little modification. For instance, the `getopt` function which is used to parse command-line options is using the global variables `optarg`, `optind`, `opterr`, and `optopt`. As explained in Sect. 7.3, the runtime library copies the values of these variables at reserved places inside the sandbox.

Other functions, *e.g.* `gethostbyname`, allocate memory on their own and return a pointer to this piece of data which is therefore not accessible to the sandboxed code. For the specific case of `gethostbyname`, the library provides the function `gethostbyname_r` which, instead of allocating memory, takes as argument a data-structure that is filled by the function. In our case, we pass as argument a sandbox allocated piece of memory. This does not solve our problem entirely as inner pointers may still point outside the sandbox. To cope with this issue, we perform a deep copy of the relevant piece of data inside the sandbox.

A last issue is that the video memory is shared between the application and the X server using the system call `shmat`. Fortunately, the libC provides the relevant flags to

[3] https://disenchant.net/git/tyrquake.git.

bind shared memory at a specific address. Hence, we were able to allocate it inside the sandbox thus allowing a seamless communication with the X server. After these modifications, the sandboxed Quake runs without noticeable slowdown which is encouraging and an indication of the good overall performance of our sandboxing technique. In the following, we complement this with a more precise runtime evaluation for the CompCert benchmarks.

8.2 PSFI Overhead: Impact of Sandboxing Primitives

Next, we compare the efficiency of a standard masking primitive (Sect. 4.1) with a specialised version for 32-bit sandboxes (Sect. 4.2).

Figure 4 shows the overhead of the standard sandboxing primitive with respect to the specialised sandboxing primitive. There are 6 benchmarks for which the overhead incurred by the standard sandboxing is above 10% reaching 40% for 2 benchmarks. These cases illustrate the significant performance advantage that is sometime obtained by the specialised sandboxing. For some benchmarks, the standard sandboxing outperforms our optimised sandboxing. Yet when it does it is by a very small margin (below 3%). Overall, for the vast majority of our benchmarks, the specialised sandboxing primitive is very competitive.

In Sect. 4.1, we gave theoretical arguments for the advantage of the specialised sandboxing. Another argument comes from the fact that the specialised sandboxing is easier to optimise. First, note that the standard and the specialised sandboxing primitives are both using a bitwise mask but for different purposes. For the standard primitive, it is used to enforce that the pointer is within the sandbox bounds but also to enforce alignment constraints. For the specialised primitive, it is only used to enforce alignment constraints. Using the existing CompCert dataflow framework, we have implemented an alignment analysis that is quite effective at removing redundant alignment masks. To enable more optimisations, we explicit alignment constraints in the Cminor code program (e.g. by specifying that function arguments of a pointer type are necessarily aligned). Thus, our experimental results are explained by both the theoretical advantages given in Sect. 4.2 and the effectiveness of our alignment analysis.

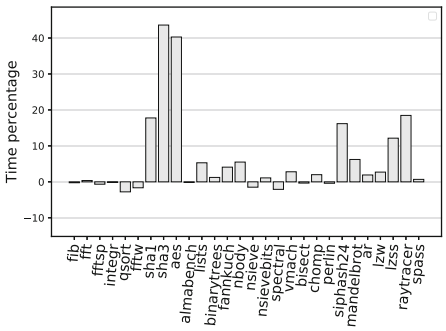


Fig. 4. Overhead of standard w.r.t specialised sandboxing

8.3 PSFI Overhead: Impact of Compiler Back-End

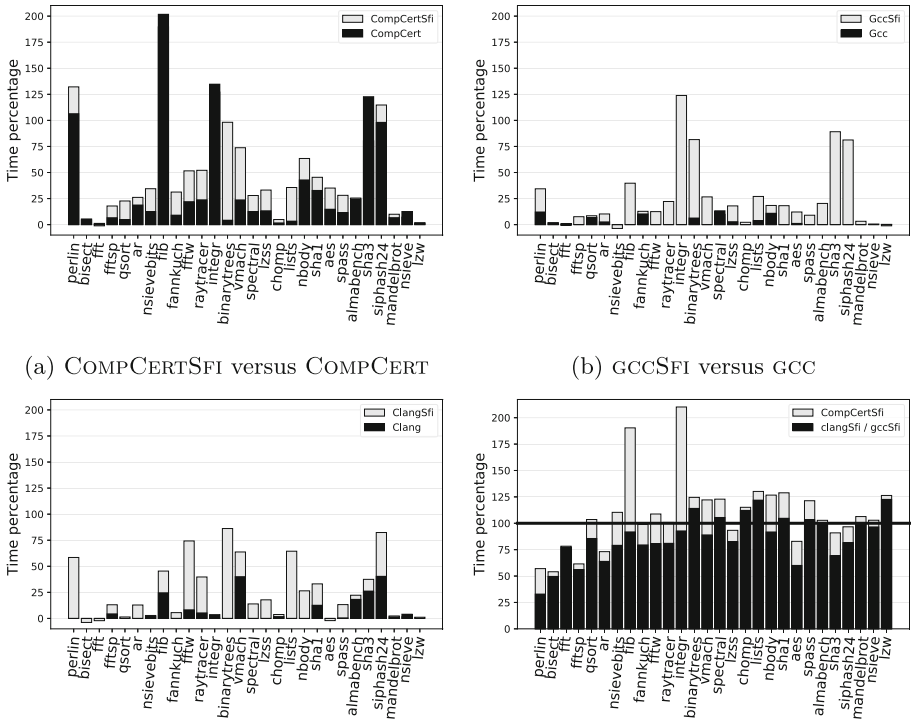
As a second experiment, we evaluate the overhead of our PSFI transformation for various compilers: CompCert, gcc and clang. CompCert is a *moderately optimising compiler* and the benchmarks run significantly faster using gcc and clang. In Fig. 5, the baseline is given by the minimum of the execution times of the three compilers without PSFI instrumentation. The black bar is the overhead of a compiler (e.g. CompCert), with respect to the baseline and the grey bar is the overhead of the same compiler but with the PSFI transformation (e.g. CompCertSfi). In order to use gcc and clang, we implement a trusted decompiler from our secured Cminor programs to Clight, a subset of C in CompCert. These Clight programs are then compiled with gcc or clang.

For a fair comparison, we should compare programs for which we actually have a reasonable security guarantee. We have a formal proof of security and safety (see Sect. 6) for the sandboxed Cminor program, and we are confident that our syntax-directed decompiler preserves this property. For CompCert, this would suffice to preserve the security of the compiled Clight code, but this is not the case for gcc and clang because of semantic discrepancies between the compilers. To limit this risk, we have set the compiler flags to instruct gcc and clang to adhere to the specificity of CompCert semantics: signed integer arithmetic is defined and so are wraps around (flag `-fwrapv`), strict aliasing is irrelevant (flag `-fno-strict-aliasing`), and floating-point arithmetic is strictly IEEE 754 compliant (flags `-frounding-math` and `-fsignaling-nans`). We also instruct the compilers to ignore any knowledge about the C library (`-fno-builtin`).

Our experimental results are shown in Fig. 5. In Fig. 5a, we have the overhead of CompCert and CompCertSfi. The overhead of CompCert over gcc and clang is expected and corroborates existing results[4]. For 10% of the benchmarks, the overhead CompCertSfi over CompCert is negligible and sometimes the PSFI transformation even improves performance. Those are programs for which the PSFI transformation introduces few masking operations, if any. For 41% of the benchmarks, the overhead is below 10% and can be considered, for most applications, a reasonable efficiency/security trade-off. For all the other benchmarks except `binarytrees` and `vmach`, the overhead is below 25%. The two remaining benchmarks have a significant overhead reaching 82% for `binarytrees`. This corresponds to programs which are memory intensive and where sandboxing cannot be optimised.

In Fig. 5b and c, we perform the same experiments but with gcc and clang. The results have some similarities but also have visible differences. For about 60% of the benchmarks the overhead is below 20%. Moreover, for both compilers, the average overhead is similar: 22% for gccSfi and 24% for clangSfi. Yet, on average gccSfi makes a better job at optimising our benchmarks and best clangSfi for about 75% of the benchmarks. For the rest of the benchmarks, we observe a significant overhead, up to 20%, indicating that the PSFI transformation hinders certain aggressive optimisations. The results also seem to indicate that optimisations are fragile as the overhead is not always consistent across compilers. The case of the `integr` benchmark is particularly striking because it runs with negligible overhead for clangSfi but exhibits the worst case overhead for gccSfi. The `integr` program is using a function pointer inside a loop and we suspect that gccSfi, unlike clangSfi, fails to optimise the program due to the inserted trampoline code. Though less striking, the benchmarks `fftw` and `raytracer` follow the opposite trend; these are programs where the overhead of clangSfi is much higher than gccSfi.

[4] http://compcert.inria.fr/compcert-C.html#perfs.



(a) CompCertSfi versus CompCert

(b) gccSfi versus gcc

(c) clangSfi versus clang

(d) PSFI versus (P)NaCl

Fig. 5. Overhead of PSFI:CompCert, clang, gcc, (P)NaCl

8.4 PSFI Versus (P)NaCl

We also compare our compiler-based SFI approach with (P)NaCl [30], which to our knowledge is one of the most mature implementations of SFI. Figure 5d shows the overhead of CompCertSfi, gccSfi, clangSfi with respect to (P)NaCl. The baseline is given by the best among NaCl and PNaCl. The best of clangSfi and gccSfi is given in dark gray and CompCertSfi is given in light grey.

We first analyse the results of CompCertSfi. Our benchmarks are ordered by increasing runtime. The first 5 benchmarks have a runtime below one second. They are not representative of the performance of both approaches but only illustrate the fact that (P)NaCl has a startup penalty due to the verification of the binary and the setup of the sandbox. The overhead peaks above 75% for two programs (i.e., `fib` and `integr`). As the PSFI transformation keeps `fib` unmodified and only inserts a trampoline call in `integr`, these programs only highlight the limited optimisations performed by CompCert. Of the remaining benchmarks, 40% of them run faster or have similar speed with CompCertSfi. For those benchmarks, the average overhead of CompCertSfi w.r.t (P)NaCl is around 9%. Except for a few programs whose overhead skyrockets due to CompCert not being specialised for speed, we can say that CompCertSfi performance is comparable to (P)NaCl, having programs with better speed in both sides and a large number having similar results.

We also matched gccSfi/clangSfi against (P)NaCl to compare the impact on performance of more aggressive optimisations. Here 60% of the programs are faster with gccSfi/clangSfi. Among the remaining programs, `lzw` and `chomp` are programs for which the (P)NaCl code runs faster than the optimised gcc clang code without the PSFI transformation. As (P)NaCl is based on clang, more investigation is needed to understand this paradox that may be explained by code running outside the sandbox *i.e.* the trusted runtime library. Among the remaining benchmarks, `binarytrees` and `lists` still show a noticeable overhead. Those are recursive micro-benchmarks for which our PSFI is costly (see Fig. 5). For `lists`, 99% of the time is spent in a tight loop where only a single address is masked. For `binarytrees`, 70% of the time is spent in the runtime code of `malloc` and `free` and therefore this highlights the fact that our implementation is less efficient than the (P)NaCl counterpart. Overall these results indicate that our implementation of SFI is competitive with (P)NaCl, given similar compilers. Furthermore speed can be improved with more sandbox-dedicated optimisations; these would be harder for (P)NaCl to check.

9 Related Work

Since Wahbe *et al.* [35] proposed their initial technique for SFI, there has been a number of proposals for efficiently confining untrusted software to a memory sandbox (see [23, 24,31,32,34,37,39]). One of the most prominent is Google's Native Client (NaCl) [37], which provides an infrastructure for executing untrusted native code in a web browser. NaCl was specifically targeted at executing computation-intensive applications without incurring a performance penalty. Certain features (in particular self-modifying code) were ruled out. These restrictions were addressed in a subsequent work [3].

RockSalt [24] is an SFI verifier for x86 code which has been developed and formally verified with the proof assistant Coq. The major contribution of RockSalt is to provide a formal model of the x86 architecture, from which it is possible to extract a decoder for a subset of the very rich set of x86 instructions, and build a verifier for the NaCl sandbox policy. Their experiments show that the formally verified checker performs marginally better than the NaCl verifier. In comparison, our approach avoids the complexities of the x86 instruction set by relying on the CompCert compiler back-end to produce binaries whose adherence to the sandbox policy is guaranteed by a combination of a sandbox verification at a higher level (Cminor) and the CompCert's correctness theorem.

ARMor [39] is using the binary rewriter Diablo [28] to implement SFI for ARM processors. Using an untrusted program analysis, a proof of SFI safety is automatically constructed using the HOL theorem prover. ARMor was tested with some programs of the MiBench benchmark [11], namely BitCount and StringSearch. These programs required 2.5 and 8 h respectively to prove the memory safety and control-flow integrity of the executables, which means that the approach is not practically viable as it is.

Kroll *et al.* [16] proposed PSFI as an alternative methodology to the standard, verification-based SFI. In PSFI, the sandbox is built by inserting the necessary masking instructions during compilation. This means that the correctness of the transformation can be argued at an intermediate stage in the compilation where the program representation retains a high-level structure. Our work extends the seminal proposal in a number of ways that we detail below. Unlike Kroll *et al.*, we exclude from the TCB the masking primitive and the trampoline mechanism for calling external functions. In our implementation, these crucial components are written entirely in Cminor and

proved correct without introducing trusted, unproved, code. Kroll *et al.* sketch a proof of safety but do not identify the issue of pointer arithmetic. To sidestep the semantics limitation of pointer arithmetic, we introduce a compile-time encoding of pointer as integers. This transformation is instrumental for our Coq verified proof of safety, which itself is mandatory to transfer security down to assembly.

Since the seminal work of Norrish [27], several works propose formal semantics of the C language [8,12,15]. All these share the limitations of CompCert with respect to pointer arithmetic. Recent works specifically aim at providing a more defined semantics for pointers. The proposal of Besson *et al.* [4] is able to cope with most existing low-level pointer manipulations and has been ported to CompCert [5,6]. Yet, it has nonetheless limitations and the design of our PSFI transformation would not benefit from the increased expressiveness. The semantics of Kang *et al.* [14] is more permissive because, after a cast, a pointer is indistinguishable from an integer value. To our knowledge, their semantics has not been ported to the CompCert compiler. Our SFI transformation has the advantage of being compatible with the existing semantics of CompCert with the caveat that pointers needs to be explicitly compiled into integers.

10 Conclusion

We have presented CompCertSfi, a formally verified implementation of Software Fault Isolation based on the CompCert compiler. Our approach provides security guarantees at runtime when the source code may be malicious or has security vulnerabilities but the build process is trusted. This is typically the case when a final product is built using code originating from multiple third parties. Our work shows that it is possible to perform security-enhancing compilation that is both formally verified and competitive with existing approaches in terms of efficiency. CompCertSfi does not rely on *a posteriori* binary verification for guaranteeing security, and hence has a reduced TCB compared to traditional SFI solutions. The reduction in TCB is obtained through a formal, machine-checked proof of the fact that the security guaranteed by our SFI transformation in the compiler front-end, still holds at the assembly level. Key to achieving this property has been to fine-tune the transformation (and in particular its pointer manipulations) to ensure that the secured program has a well-defined semantics.

The impact of SFI has been evaluated on a series of benchmarks, showing that the transformed code can in a few cases be more efficient, and that the average runtime overhead incurred is about 9%. We have evaluated the impact of back-end optimisation on the transformed code on three different compilers. The gains vary, with clang being more efficient than CompCert and gcc, and CompCert being slightly more efficient than gcc. The experiments show that CompCertSfi combined with an aggressive back-end optimiser can sometimes achieve performances superior to Native Client implementations. In addition, there is still room for further optimisation of the generated code. We have observed that existing optimisations are sometimes hindered by our SFI transformation, so we gain by having more optimisation before the SFI transformation. We also intend to investigate optimisations for removing redundant sandboxing operations and in particular hoisting sandboxing outside loops.

References

1. Supplementary material. https://www.irisa.fr/celtique/ext/compcertsfi
2. Andronick, J., Chetali, B., Ly, O.: Using Coq to verify Java Card^{TM} applet isolation properties. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 335–351. Springer, Heidelberg (2003). https://doi.org/10.1007/10930755_22
3. Ansel, J., et al.: Language-independent sandboxing of just-in-time compilation and self-modifying code. In: PLDI, pp. 355–366 (2011)
4. Besson, F., Blazy, S., Wilke, P.: A precise and abstract memory model for C using symbolic values. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 449–468. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_24
5. Besson, F., Blazy, S., Wilke, P.: CompCertS: a memory-aware verified C compiler using pointer as integer semantics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 81–97. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_6
6. Besson, F., Blazy, S., Wilke, P.: A verified CompCert front-end for a memory model supporting pointer arithmetic and uninitialised data. J. Autom. Reasoning (2018, accepted for publication)
7. Besson, F., de Grenier de Latour, T., Jensen, T.P.: Interfaces for stack inspection. J. Funct. Program. **15**(2), 179–217 (2005)
8. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL. ACM (2012)
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
10. Guanciale, R., Nemati, H., Dam, M., Baumann, C.: Provably secure memory isolation for Linux on ARM. J. Comput. Secur. **24**(6), 793–837 (2016)
11. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: a free, commercially representative embedded benchmark suite, pp. 3–14. Institute of Electrical and Electronics Engineers Inc., United States (2001)
12. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: PLDI, pp. 336–345. ACM, June 2015
13. ISO: ISO C Standard 1999. Technical report (1999)
14. Kang, J., Kim, Y., Hur, C., Dreyer, D., Vafeiadis, V.: Lightweight verification of separate compilation. In: POPL, pp. 178–190. ACM (2016)
15. Krebbers, R.: An operational and axiomatic semantics for non-determinism and sequence points in C. In: POPL. ACM (2014)
16. Kroll, J.A., Stewart, G., Appel, A.W.: Portable software fault isolation. In: CSF, pp. 18–32. IEEE (2014)
17. Larus, J.R., Hunt, G.C.: The singularity system. Commun. ACM **53**(8), 72–79 (2010)
18. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
19. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
20. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model. In: Program Logics for Certified Compilers. Cambridge University Press (2014)

21. Leroy, X., Rouaix, F.: Security properties of typed applets. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming, Security Issues for Mobile and Distributed Objects. LNCS, vol. 1603, pp. 147–182. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48749-2_7
22. The Coq development team: The Coq proof assistant reference manual (2017). http://coq.inria.fr, version 8.7
23. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: Proceedings of the 15th Conference on USENIX Security Symposium, USENIX-SS 2006, vol. 15. USENIX Association (2006)
24. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: PLDI, pp. 395–404. ACM (2012)
25. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119. ACM Press (1997)
26. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: OSDI, pp. 229–243. ACM (1996)
27. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
28. Put, L.V., Chanet, D., Bus, B.D., Sutter, B.D., Bosschere, K.D.: DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: In IEEE International Symposium On Signal Processing And Information Technology (2005)
29. Richards, G., Hammer, C., Nardelli, F.Z., Jagannathan, S., Vitek, J.: Flexible access control for JavaScript. In: OOPSLA, pp. 305–322. ACM (2013)
30. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures. In: 19th USENIX Security Symposium, pp. 1–12. USENIX Association (2010)
31. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 1. USENIX Association (2010)
32. Shu, R., et al.: A study of security isolation techniques. ACM Comput. Surv. **49**(3), 50:1–50:37 (2016)
33. Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: EuroS&P, pp. 1–15. IEEE (2018)
34. Sinha, R., et al.: A design and verification methodology for secure isolated regions. In: PLDI, pp. 665–681. ACM (2016)
35. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: SOSP, pp. 203–216. ACM (1993)
36. Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M.: Undefined behavior: what happened to my code? In: APSYS (2012)
37. Yee, B., et al.: Native client: a sandbox for portable, untrusted x86 native code. In: S&P, pp. 79–93. IEEE (2009)
38. Yee, B., et al.: Native client: a sandbox for portable, untrusted x86 native code. Commun. ACM **53**(1), 91–99 (2010)
39. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: ARMor: fully verified software fault isolation. In: EMSOFT, pp. 289–298. ACM (2011)