Modular Software Fault Isolation as Abstract Interpretation

Frédéric Besson, Thomas Jensen, and Julien Lepiller

Inria, Univ Rennes, CNRS, IRISA

Abstract. Software Fault Isolation (SFI) consists in transforming untrusted code so that it runs within a specific address space, (called the sandbox) and verifying at load-time that the binary code does indeed stay inside the sandbox. Security is guaranteed solely by the SFI verifier whose correctness therefore becomes crucial. Existing verifiers enforce a very rigid, almost syntactic policy where every memory access and every control-flow transfer must be preceded by a sandboxing instruction sequence, and where calls outside the sandbox must implement a sophisticated protocol based on a shadow stack. We propose to define SFI as a defensive semantics, with the purpose of deriving semantically sound verifiers that admit flexible and efficient implementations of SFI. We derive an executable analyser, that works on a per-function basis, which ensures that the defensive semantics does not go wrong, and hence that the code is well isolated. Experiments show that our analyser exhibits the desired flexibility: it validates correctly sandboxed code, it catches code breaking the SFI policy, and it can validate programs where redundant instrumentations are optimised away.

1 Introduction

A fundamental challenge in system security is to share computing resources and run programs from various level of trusts, some untrusted or even malicious, on the same host machine. In this context, it is desirable to isolate the different programs, limit their interactions and ensure that, whatever the behaviour of imported code, the security of the host machine cannot be compromised. There exist many isolation mechanisms available at the hardware, virtual machine or operating system level. In this paper, we consider Software Fault Isolation (SFI), an isolation mechanism pioneered by Wahbe *et al.* [15] and further developed in Google's Native Client (NaCl) [16, 13] and others. SFI is a flexible and lightweight isolation mechanism which does not rely on hardware or operating system support. Instead, it relies on a static, untrusted, program instrumentation that is validated at load-time by a trusted binary verifier. Compared to other isolation mechanisms, SFI allows the safe execution of trusted and untrusted code within the same address space, thus avoiding costly context switches with kernel code.

The general SFI architecture is shown in Fig. 1 together with some typical sandboxing code. The SFI transformation is performed at compile-time. It
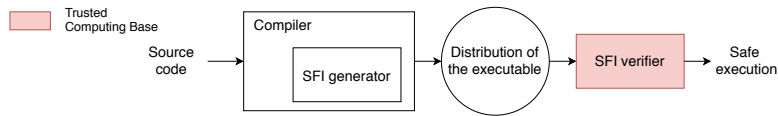
```
static  inline  int  sandbox ( int  p )  { return  (& sfi  +  ( p  &  0b11111111 ))  ;  }
```

Fig. 1: SFI chain with typical sandboxing code

instruments every memory access so that it is performed inside the memory sandbox. To ensure that a pointer p is within the sandbox variable sfi that is 2^8 bytes aligned and 2^8 bytes wide, the code increments the base address sfi of the sandbox with the 8 least significant bits of the pointer p extracted by masking p using a bitwise &. Control-flow transfers are instrumented so that the code does not jump outside the code sandbox. At load-time, a binary verifier rejects code that is not correctly instrumented. From a security standpoint, only the binary verifier is part of the Trusted Computing Base (TCB). State-of-the-art SFI verifiers trade precision for simplicity and speed, and only perform a linear scan of the binary code. Therefore, the verifiers enforce very strong sufficient conditions for isolation. This has the side-effect that the SFI transformation is performed late in the compiler backend and that the isolated code cannot be optimised. The verifiers perform very local reasoning, and hence cannot verify that function calls, especially between trusted and untrusted code, abide to calling conventions. As a result, trusted code needs to implement a specific protocol for parameter passing and set up its own private run-time stack. This requires low-level platform-specific support and, most notably, increases the run-time overhead of context switches between untrusted and trusted code.

In this paper, we propose a relaxed definition of SFI where trusted and untrusted code may share the same runtime stack but must still respect the isolation properties of the sandbox and abide to calling conventions. Based on this definition, we define an intra-procedural binary verifier which enforces isolation. The verifier implements a static analysis and is using a weakly relational domain in order to verify that calling conventions are satisfied. A difficulty is to ensure that the isolation property holds even in the presence of stack overflow. This is done by ensuring that all stack overflows are caught by so-called *guard zones*, placed at both ends of the stack. The binary verifier is more flexible than state-of-the-art SFI verifiers and, in particular, is able to validate code where redundant sandboxing instrumentations are optimised away by compiler passes. Our contributions can therefore be phrased as follows:

- A defensive semantics which formalises a relaxed Software Fault Isolation property where the runtime stack is safely shared between trusted and untrusted code.
- An intra-procedural abstraction which ensures that the defensive semantics cannot go wrong.
- An executable binary verifier, working on a per-function basis, that is more flexible than state-of-the-art binary verifiers for SFI.

The rest of the paper is organised as follows. In Section 2, we define our SFI property by means of a defensive, instrumented semantics. To enable a modular verification, we present in Section 3 an intra-procedural abstraction of the defensive semantics. It is further abstracted in Section 4 into an executable binary verifier. Section 5 presents our experiments based on the BINCAT [2] binary analysis framework. We present related work in Section 6 and conclude in Section 7.

2 Software Fault Isolation as a Defensive Semantics

We define SFI and its sandbox property operationally, as a defensive semantics which includes a series of additional (dynamic) verifications. These dynamic checks express what it means for code to be properly sandboxed. Later, we define a static analysis for guaranteeing that the dynamic verifications will not fail at run-time, and hence that the code respects the SFI property.

2.1 Intermediate Language

We define our semantics on an intermediate representation (IR) obtained by disassembling the binary. In this approach, each binary instruction is typically translated into a sequence of instructions of the IR. For instance, for x86, a simple arithmetic operation has the side-effect of setting various flags e.g. the carry or overflow flag. For simplicity, we also assume that the IR only manipulates 32-bits values. The abstract syntax of the instructions is given below:

$$e ::= r \mid n \mid e_1 \bowtie e_2$$
$$i ::= r := e \mid [e_1] = e_2 \mid r = [e] \mid \textbf{jmpif } e_1 \ e_2 \mid \textbf{call } e \mid \textbf{ret } e \mid \textbf{hlt}$$

The language features expressions e made of registers r, numeric constants n and binary operators $\bowtie$. Binary operators range over typical arithmetic operators e.g. $+$, $\times$, bitwise operators e.g. xor and logical operators e.g. $<$. An instruction i consists of assigning an expression to a register ($r = e$); storing in memory the value e_2 at the address e_1 ($[e_1] = e_2$); loading in register r the value stored at address e ($r = [e]$). A conditional jump **jmpif** e_1 e_2 jumps to the computed address e_2 if the condition e_1 holds ($e_1 \neq 0$). The instruction **call** e is equivalent to the computed jump **jmpif** 1 e but identifies a function call; **ret** e is also equivalent to a computed jump but identifies a function return. The instruction **halt** immediately stops the program.

The operational semantics of the IR operates over a state $\langle \rho, \mu, \iota \rangle$ where ρ is an environment ($Env = Reg \to \mathbb{B}_{32}$), μ is the whole memory of the process and ι is the current instruction pointer. The memory is divided into regions and each region is granted access rights among **r**ead, **w**rite and e**x**ecute that are checked for by the semantics. For instance, before reading in memory, we check that the address has the read permission **r**. The rules that give the semantics of each instruction are fairly standard and are given in Appendix A.

2.2 Semantic Domains

Our defensive semantics makes use of several semantic domains that we describe below. Our notations are fairly standard: the set $\mathbb{B} = \{0, 1\}$ is the set of booleans and we write $\mathbb{B}_{32}$ for $\mathbb{B}^{32}$ which reads *bitvector of size 32*. A stack frame is a pair $\langle bp, \phi \rangle \in \mathbb{B}_{32} \times \mathbb{B}_{32}^*$ where bp represents the base pointer of the stack frame and ϕ is a list of 32-bits values modelling the content of the stack frame. The semantics is using several architecture-dependent constants. The constant d_0 is the base address of the sandbox, the constant s_0 is the top address of the stack, and the maximum size of a stack frame is f_s.

In order to detect stack overflows at runtime, the runtime stack is surrounded by so-called *guard zones*. The concept of guard zone was already present in the original work on SFI [15]. Semantically, this is modelled as memory regions which have no access rights. As a result, accesses within the guard zones are trapped, by letting the execution enter a specific "crash state" $\blacksquare$ where it stays forever. In the following, $\mathrm{GZ}_\top$ is the size of the guard zone at the top of the stack and $\mathrm{GZ}_\bot$ is the size of the guard zone at the bottom of the stack. The guard zones are part of the stack. A call stack $CS = (\mathbb{B}_{32} \times \mathbb{B}_{32}^*)^*$ is a list of stack frames such that the successive base addresses are decreasing (the stack grows downward). The length of the (intermediate) stack frames is given by the difference between two successive base pointers. A call stack is immutable but the content of the call stack can be read. This is modelled by the following judgement $cs \vdash_a v$ which reads *the call stack cs contains the value v at address a*.

$$\frac{a \leq bp \quad \phi(bp - a) = \lfloor v \rfloor}{cs :: \langle bp, \phi \rangle \vdash_a v} \qquad \frac{a > bp \quad cs \vdash_a v}{cs :: \langle bp, \phi \rangle \vdash_a v} .$$

2.3 Defensive Semantics

At binary level, the runtime stack and the code segment are no different from any other part of memory. Our defensive semantics must therefore explicitly enforce a stack discipline and ensure that function boundaries are respected. The program text is located in memory. For this purpose, we identify a set of addresses $Code \subset \mathbb{B}_{32}$ that correspond to the program code. Given an address i, the function $instr(i)$ checks that $i \in Code$ and returns the instruction stored at address i. Moreover, we assume given a set $\mathcal{F} \subseteq Code$ of function entry points, and a set $\mathcal{T} \subseteq \mathbb{B}_{32}$ of trusted functions that form the only authorized entry points of the trusted library.

A semantic derivation occurs in a context $\langle cs, bp, \rho_i \rangle \in CS \times \mathbb{B}_{32} \times Env$ where cs is the call stack, bp is the base pointer of the current frame and ρ_i is the environment at the function entry. A judgement is of the form $\Gamma \vdash s \to s'$ where Γ is an inter-procedural context and s, s' are either intra-procedural states $s, s' \in State = Env \times \mathbb{B}_{32}^* \times \mathbb{B}_{32}^* \times \mathbb{B} \times \mathbb{B}_{32}$ or the crash state $\blacksquare$. A state $\langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \in State$ is made of an environment ρ mapping registers to values, a public data segment δ i.e. the sandbox, a stack frame ϕ, a boolean β which tells whether a write has occurred in the current frame, and the current instruction pointer ι. (We write ι^+ for the pointer to the next instruction.) The semantics

also ensures that there is no overlap between the call stack (including the current stack frame), the code, and the data segment.

The semantics rules are found in Fig. 2. The ASSIGN rule assigns a new value

$$\text{ASSIGN}\ \frac{instr(\iota) = \lfloor r = e \rfloor}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho[r \mapsto \llbracket e \rrbracket_\rho], \delta, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{STOREDATA}\ \frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = \llbracket e_1 \rrbracket_\rho - d_0 \quad 0 \leq o <\mid \delta \mid}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho, \delta[o \mapsto \llbracket e_2 \rrbracket_\rho], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{STOREFRAME}\ \frac{\begin{array}{c} instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = bp - \llbracket e_1 \rrbracket_\rho \\ 0 \leq o <\mid \phi \mid -\text{GZ}_\perp \quad bp - o \leq s_0 - \text{GZ}_\top \end{array}}{\langle cs, bp, R \rangle \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho, \delta, \phi[o \mapsto \llbracket e_2 \rrbracket_\rho]^{(1)}, \iota^+ \rangle}$$

$$\text{LDSTCRASH}\ \frac{\begin{array}{c} instr(\iota) = \lfloor [e_1] = e_2 \rfloor \vee instr(\iota) = \lfloor [r = [e_1]] \rfloor \quad \llbracket e_1 \rrbracket_\rho = a \\ (s_0 - \text{GZ}_\top < a \leq s_0) \vee (bp- \mid \phi \mid < a \leq bp- \mid \phi \mid +\text{GZ}_\perp) \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \blacksquare}$$

$$\text{LOADDATA}\ \frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = d_0 + o \quad 0 \leq o <\mid \delta \mid}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho[r \mapsto \delta(o)], \delta, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{LOADSTACK}\ \frac{\begin{array}{c} instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = a \quad cs :: \langle bp, \phi \rangle \vdash_a v \\ bp- \mid \phi \mid +\text{GZ}_\perp < a \leq s_0 - \text{GZ}_\top \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho[r \mapsto v], \delta, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{CALL}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \textbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{F} \quad \rho(\textbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \\ o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad instr(\iota') = \lfloor \textbf{ret}\ e' \rfloor \quad \llbracket e' \rrbracket_{\rho'} = \iota^+ \\ \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle \vdash \langle \rho, \delta, \phi_2^{(0)}, f \rangle \longrightarrow^* \langle \rho', \delta', \phi_2'^{(\beta)}, \iota' \rangle \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle \longrightarrow \langle \rho', \delta', \phi_1 \cdot \phi_2'^{(1)}, \iota^+ \rangle}$$

$$\text{CALLTRUST}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \textbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{T} \\ \rho(\textbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle \longrightarrow \langle \rho', \delta', \phi_1 \cdot \phi_2'^{(1)}, \iota^+ \rangle}$$

$$\text{CONT}\ \frac{instr(\iota) = \lfloor \textbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = 0}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho, \delta, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{JUMP}\ \frac{instr(\iota) = \lfloor \textbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho \neq 0 \quad \llbracket e_2 \rrbracket_\rho \in Code}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \langle \rho, \delta, \phi^{(\beta)}, \llbracket e_2 \rrbracket_\rho \rangle}$$

$$\text{HALT}\ \frac{instr(\iota) = \lfloor \textbf{hlt} \rfloor}{\Gamma \vdash \langle \rho, \delta, \phi^{(\beta)}, \iota \rangle \longrightarrow \blacksquare} \qquad \text{CRASH}\ \frac{}{\Gamma \vdash \blacksquare \longrightarrow \blacksquare}$$

Fig. 2: Defensive semantics

to a register. This is always possible without violating the SFI property and no extra check is needed. The rule STOREDATA describes the execution of the statement $[e_1] = e_2$ for the case where e_1 evaluates to a memory address within the sandbox. The value of e_1 is computed and the start address of the data segment (the sandbox) d_0 is subtracted from it to obtain an offset o into the

data segment. It is then verified that this offset is indeed smaller than the size of the data segment. If this verification succeeds, the location at offset o in the sandbox is updated with the value of e_2.

The rule STOREFRAME similarly makes the checks necessary for storing securely into the run-time stack. Here, the value of e_1 is supposed to be a valid reference into the current stack which starts at the address designated by the base pointer bp. Because the stack grows towards smaller addresses, the relative offset o into the current stack frame is computed as $bp - [\![e_1]\!]_\rho$. In order for the store to proceed normally, this offset must point into that part of the stack frame that is *not* making up the guard zone ($0 \le o <| \phi | -\text{GZ}_\perp$). It is also checked that the offset does not point into the guard zone at the beginning of the stack segment ($bp - o \le s_0 - \text{GZ}_\top$). This rule also sets β to 1, which has the side-effect of ensuring the current base pointer is above the guard zone (there is some space to write to). The rule LDSTCRASH describes what happens on an attempt to write into or read from one of the guard zones. In that case, the program transits to the crash state ■ and stays there forever due to rule CRASH.

The two rules LOADDATA, LOADSTACK describe how data are read from the data segment and the run-time stack. Reading from the data segment uses verification similar to storing into it. Loading from the stack is, however, slightly different in that our version of SFI allow reads from all of the stack frames, and not just the current frame. This allows e.g. functions to read their arguments. It is still verified that the access does not fall in the guard zones, using checks similar to STOREFRAME.

The rule CALL for the function call instruction **call** e first verifies that the value $[\![e]\!]_\rho$ belongs to the set of function entry points $\mathcal{F}$. The current stack frame is divided into two parts $\phi_1 \cdot \phi_2$ where ϕ_1 is local data of the caller and ϕ_2 is the new stack frame for the function call, which starts at the address contained in register **esp**. The offset o between the start of the old stack frame and the new is verified to be smaller than the maximal frame size f_s. Because it is checked that $\beta = 1$, i.e. the frame has already been written to in this function, this check has the side-effect to ensure bp is above the guard zone $\text{GZ}_\perp$ and therefore the new bp is still at least f_s above the bottom of the stack. Note that enforcing a write before calls is not restrictive as writes are generated by compilers before function calls in any architecture. The actual method call is modelled as an execution starting at address f with the same environment ρ, the same data segment δ, and a stack frame $\phi_2^{(0)}$ where the 0 indicates that the frame has not yet been written into. The end of the call is identified by the execution reaching a **ret** e' instruction. The value of $[\![e']\!]_{\rho'}$ is verified to be the return address using the architecture-dependent predicate *isret*. The return address is the next instruction to execute. The semantics verifies that callee-saved registers are restored after the function call. For instance on X86 assembly, registers **esp**, **ebx** etc are saved. For this same architecture, *isret* checks that the return address has indeed been pushed to the call stack. Calling a trusted function is modelled with the rule CALLTRUST. This rule follows the same pattern as the rule for ordinary calls, except that the trusted call is allowed to modify the sandbox

data but should leave the callee's stack frame unchanged. We model this as a non-deterministic rule that can return any δ' in its resulting state.

The rules CONT and JUMP model the instruction **jmpif** e_1 e_2 for conditional jumps to a computed address. If the condition e_1 evaluates to zero, execution continues with the next instruction. Otherwise, the value $[\![e_2]\!]_\rho$ is computed and it is verified that this new jump target is in the code block *Code*. Finally, we use the (secure) crash state ■ to model program termination in the rule HALT. The rule CRASH states that once in a crash state the execution stays in this state forever. This semantic sleight of hand simplifies the statement of the overall security property, which becomes essentially a progress property. Employing a specific error state would be equivalent but slightly more cumbersome.

2.4 The Sandbox Property

The side-conditions of the rules performing memory accesses ensure that the defensive semantics gets stuck when memory accesses do not respect the sandboxing property. This means that we can state our sandbox property as a simple *progress* property of the defensive semantics: as long as the semantics can progress to a new state (possibly the crash state) no security violation has occurred.

There is one obstacle to this, though: due to our *big-step* modelling of function calls, the semantics also gets stuck as soon as a function call does not terminate. In other words, all infinite loops are deemed insecure, which is clearly not what we want. To remedy this, our sandbox property is defined over the set of reachable states induced by the defensive semantics where the transition relation $\rightarrow$ is extended with a relation $\triangleright$ which for each **call** instruction explicitly adds a transition to the callee state.

$$\text{CALLACC} \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \end{array}}{\langle\langle cs, bp, \rho_i\rangle, \langle \rho, \delta, \phi_1 \cdot \phi_2^{(1)}, \iota\rangle\rangle \triangleright \langle\langle cs :: \langle bp, \phi_1\rangle, bp - o, \rho\rangle, \langle \rho, \delta, \phi_2^{(1)}, f\rangle\rangle}$$

Dually, we also add a transition stating that, for a **ret** instruction, the return state is not stuck provided that the calling conventions are respected. Since the next step is taken care of by the CALL rule, the resulting state is just a witness that the execution can proceed; we reuse for this purpose the state ■.

$$\text{RETACC} \frac{\rho_i \sim \rho' \quad isret(ret, \phi_1, \rho_i) \quad instr(\iota) = \lfloor \mathbf{ret}\ e' \rfloor \quad [\![e']\!]_{\rho'} = ret}{\langle\langle cs :: \langle bp, \phi_1\rangle, bp - o, \rho_i\rangle, \langle \rho', \delta, \phi_2'^{(\beta)}, \iota\rangle\rangle \triangleright \langle\langle cs :: \langle bp, \phi_1\rangle, bp - o, \rho_i\rangle, ■\rangle}$$

Definition 1 (Augmented defensive semantics). *The augmented defensive semantics* $\Rightarrow$ *is given by the union of the relation* $\rightarrow$ *and* $\triangleright$ *such that:*

$$\frac{\Gamma \vdash \Sigma_1 \rightarrow \Sigma_2}{\langle \Gamma, \Sigma_1\rangle \Rightarrow \langle \Gamma, \Sigma_2\rangle} \qquad \frac{\langle \Gamma_1, \Sigma_1\rangle \triangleright \langle \Gamma_2, \Sigma_2\rangle}{\langle \Gamma_1, \Sigma_1\rangle \Rightarrow \langle \Gamma_2, \Sigma_2\rangle}$$

The SFI sandbox property can then be expressed as the progress property of the augmented defensive semantics.

Definition 2 (Sandboxing as progress). *Let ι_0 be the entry point of the program and let the initial state be $\langle \Gamma_0, \Sigma_0 \rangle = \langle \langle [\langle s_0, \phi_i \rangle], s_0 - \mathrm{GZ}_\top, \rho_0 \rangle, \langle \rho, \delta, \phi^{(0)}, \iota_0 \rangle \rangle$ with $\mid \phi_i \mid = \mathrm{GZ}_\top$. The program satisfies the SFI sandbox property if the set of reachable states $Acc = \{ s \mid \langle \Gamma_0, \Sigma_0 \rangle \Rightarrow^* s \}$ satisfies $\forall s \in Acc . \exists s'. s \Rightarrow s'$.*

We write $Safe(Acc)$ if this is the case.

3 Intraprocedural semantics as an abstract interpretation

In order to derive a modular static analyser we abstract the defensive semantics into an intra-procedural semantics where the accessible states $I\text{-}Acc$ are computed for each function separately: $I\text{-}Acc = \bigcup_{f \in \mathcal{F}} I\text{-}Acc(f)$. Our intra-procedural semantics abstracts away the data region and all of the call stack, except the frame of the caller. Thus the stack component is abstracted to two small zones of the stack above and below the current base pointer, representing the frames of the caller and the callee (the currently executing function). Both are modelled by memory regions of size f_s where f_s is a chosen maximum size of these abstract stack frames. Fig. 3 illustrates the abstraction of the stack segment. The current code can write to its own frame (the W zone) and read from both frames (the R zone). The size of the guard zones are set such that the abstract frames are always contained in the stack, possibly overlapping a guard zone.
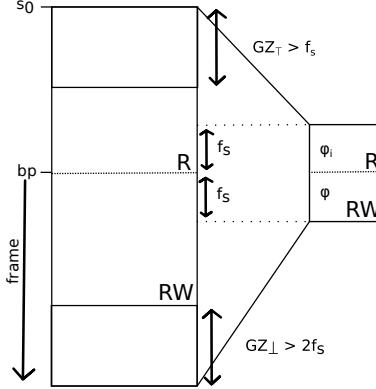

Fig. 3: Abstraction of call stack

The judgement of the intra-procedural semantics is of the form: $\Gamma \vdash s \rightarrow s'$ where the context $\Gamma = \langle \phi_i, bp, \rho_i \rangle \in Ctx^\natural$ is constant. Here, ϕ_i is the frame of the caller, bp is the base pointer and ρ_i is the initial environment. The states s and s' are either the crash state $\blacksquare$ or of the form $\langle \rho, \phi^{(\beta)}, \iota \rangle \in State^\natural$.

The intra-procedural semantics has no knowledge about where it is in the stack segment so it cannot detect stack overflows *per se*. We solve this problem by a judicious use of the guard zones. The defensive semantics enforces that a successful memory write occurs within the current stack frame before any call. This entails the semantic invariants that 1. before a function call the base pointer is always inside the stack and outside of the guard zone, 2. at function entry point, the base pointer is always inside the stack or at most f_s bytes inside the guard zone, 3. hence, there are at least f_s bytes left between bp and the end of the stack segment (including the guard zone). These invariants are formally stated by Lemma 3 that is proved in Appendix C.

Similarly, we also guarantee that the call stack is at least of size f_s, possibly overlapping the guard zone $\mathrm{GZ}_\top$. Those invariants are enough to detect both stack overflows and underflows using the intra-procedural semantics: if all the stack accesses are proved to be within the bound of f_s above and below the stack

pointer bp, the accesses are either defined in the defensive semantics or lead to a crash because the access is performed inside the guard zone.

The intra-procedural and defensive semantics are linked by the concretization function $\gamma : Ctx^\natural \times State^\natural \to \mathcal{P}((Ctx \times State) \cup \{\blacksquare\})$. The data segment is not represented in the intra-procedural semantics, so its concretization is any data segment of size d_s. The concretization constructs the call stack and the current defensive frame in such a way that, once appended to one another, they form a memory region of size s_s and f_i and f are windows of size f_s around the address pointed to by the base pointer. Formally, we have:

$$\gamma(\langle f_i, bp, \rho_i\rangle, \langle \rho, f^{(\beta)}, \iota\rangle) = \{\blacksquare\} \cup \left\{ \langle cs, bp, \rho_i\rangle, \langle \rho, \delta, \phi^{(\beta)}, \iota\rangle \,\middle|\, \begin{array}{l} f = \phi_{|_{[0,f_s-1]}} \\ f_i = \overline{cs}_{|_{[|\overline{cs}|-f_s,|\overline{cs}|]}} \end{array} \right\}$$

where $\overline{cs}$ is obtained by concatenating the different stack frames of the call stack and $\phi_{|_{[a,b]}}$ extracts the sub-list between the indexes a and b.

$$\overline{cs} = \begin{cases} [] & \text{if } cs = [] \\ \phi \cdot \overline{cs'} & \text{if } cs = \langle bp, \phi\rangle :: cs' \end{cases} \qquad \phi_{|_{[a,b]}} = \begin{cases} [] & \text{if } a > b \\ \phi(a) :: \phi_{|_{[a+1,b]}} & \text{otherwise} \end{cases}$$

Except for the handling of stack overflows and underflows, the rules for the intra-procedural semantics are fairly standard and can be found in Appendix B. When a memory component is absent from the abstraction i.e. the data region, the intra-procedural semantics non-deterministically picks a value. For the **call** instruction, the rule is similar to the defensive semantics rule CALL with the notable exception that no recursive call is made.

$$\text{FUNCALL} \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{F} \cup \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad 0 \le o < f_s \quad |\ \phi_1 \models o \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle \phi_i, bp, \rho_i\rangle \vdash \langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota\rangle \to^\natural \langle \rho', \phi_1 \cdot \phi_2'^{(1)}, \iota^+\rangle}$$

The rule FUNLDARG shows how to access the arguments of the function that are placed in the stack frame of the caller modelled by ϕ_i.

$$\text{FUNLDARG} \frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = (bp + f_s) - o \quad 0 \le o < f_s}{\langle \phi_i, bp, \rho_i\rangle \vdash \langle \rho, \phi^{(\beta)}, \iota\rangle \to^\natural \langle \rho[r \mapsto \phi_i(o)], \phi^{(\beta)}, \iota^+\rangle}$$

The base address of ϕ_i is obtained by incrementing the base pointer bp by the stack frame size f_s and checking that the offset o is in range $[0; f_s[$. The soundness of this rule exploits the fact that the defensive stack is guarded by $\text{GZ}_\top$. As a result, if FUNLDARG succeeds, either the memory access also succeeds in the defensive semantics (rule LOADSTACK) or it accesses the guard zone $\text{GZ}_\top$ and triggers a crash (rule LDSTCRASH).

For each function entry $\iota \in \mathcal{F}$, the initial states $Init(f) \subseteq Ctx^\natural \times State^\natural$ are defined by: $Init(\iota) = \left\{ \langle\langle \phi_i, bp, \rho\rangle, \langle \rho, \phi^{(0)}, \iota\rangle\rangle \,\middle|\, \begin{array}{l} |\ \phi \models |\ \phi_i \models f_s \wedge \\ bp > s_0 - s_s + \text{GZ}_\perp - f_s \end{array} \right\}$. As we

already discussed, the frames ϕ and ϕ_i have length f_s. At the function start, the environments of the caller and the callee are the same; the base pointer is so that there is below it a stack frame of size at least f_s and no memory write has been performed on the current frame ϕ. For a given function entry point $f \in \mathcal{F}$, the reachable states are defined as

$$I\text{-}Acc(f) = \{(\Gamma, s) \mid \Gamma \vdash s_0 \rightarrow^* s \wedge (\Gamma, s_0) \in Init(f)\}.$$

The intra-procedural semantics is also defensive and gets stuck when abstract verification conditions are not met.

Definition 3 (Intra-procedural progress). *The intra-procedural states are safe (written $I\text{-}Safe(I\text{-}Acc)$) iff $\forall f \in \mathcal{F}, \forall (\Gamma, s) \in I\text{-}Acc(f).\exists s'.\Gamma \vdash s \rightarrow s'$.*

The checked conditions are sufficient (but may not be necessary). For instance, the intra-procedural semantics gets stuck when an access is performed outside the bound of the current stack frame ϕ. However, because ϕ only models a prefix of the frame of the defensive semantics, the defensive semantics may not be stuck. As a result, the usual result $Acc \subseteq \gamma(I\text{-}Acc)$ does not hold. Instead, we have Lemma 1 stating that if the intra-procedural semantics is not stuck, it abstracts the defensive semantics and ensures that the accessible states of the defensive semantics are safe.

Lemma 1 (Correctness of the Intra-procedural semantics).

$$I\text{-}Safe(I\text{-}Acc) \Rightarrow Acc \subseteq \gamma(I\text{-}Acc) \wedge Safe(Acc).$$

The proof can be found in Appendix C.

4 A Static SFI Analysis

To get a modular executable verifier, we abstract further the intra-procedural semantics. The verifier needs to track numeric values used as addresses, in order to guarantee that memory accesses are within the sandbox or within the current stack frame, hence we need domains tailored for such uses of numeric values. The verifier also needs to gather some input-output relational information about the registers and verify that, at the end of the functions, callee-saved registers are restored to their initial values.

4.1 Abstract Domains

As pioneered by Balakrishnan and Reps [1], we perform a Value Set Analysis (VSA) where abstract locations (*a-locs*) are DATA for the sandbox region and CODE for the code region. We also introduce an abstract location for the function return RET, the base pointer BP and for each register e.g. EAX, EBX. To model purely numeric data, we have a dedicated *a-locs* ZERO with value 0:

$$\text{a-locs} = \{\text{ZERO}, \text{DATA}, \text{CODE}, \text{RET}, \text{BP}, \text{EAX}, \text{EBX}, \ldots\}.$$

a-locs are equipped with an arbitrary non-relational numeric domain. The abstract value domain $\mathbb{B}_{32}^\sharp$ is therefore a pair (L, o) made of an abstract location L and a numeric abstraction $o \in D^\sharp$. For each concrete operation $\diamond$ on values, the transfer function on abstract (L, o)-values is using the corresponding operation $\diamond^\sharp$ of the abstract domain. For instance, for addition and subtraction, we get:

$$(L, o_1) +^\sharp (\textsc{zero}, o_2) = (L, o_1 +^\sharp o_2) \quad (\textsc{zero}, o_1) +^\sharp (L, o_2) = (L, o_1 +^\sharp o_2)$$
$$(L, o_1) -^\sharp (\textsc{zero}, o_2) = (L, o_1 -^\sharp o_2) \quad (L, o_1) -^\sharp (L, o_2) = (\textsc{zero}, o_1 -^\sharp o_2)$$

When symbolic computations are not possible, it is always possible to abstract (L, o) by $(\textsc{zero}, \top)$ and use numeric transfer functions. As the usual sandboxing technique consists in masking an address using a bitwise & ($[e_1] := e_2 \rightsquigarrow [d_0 + e_1 \& 1^k] := e_2$) [1] we opt, in our implementation, for the bitfield domain [11].

The abstract machine state at a program point is the product of an abstract environment $Env^\sharp$, an abstract frame $Frame^\sharp$, and a code pointer $\mathbb{B}_{32}$.

$$Env^\sharp = Reg \to \mathbb{B}_{32}^\sharp \quad Frame^\sharp = (\mathbb{B}_{32}^\sharp)^{f_s} \times \mathbb{B} \quad State^\sharp = Env^\sharp \times Frame^\sharp \times \mathbb{B}_{32}.$$

The abstract frame is annotated by a boolean indicating whether a memory write has definitively occurred in the stack frame.

The concretization function is parametrised by a mapping $\lambda : \text{a-locs} \to \mathbb{B}_{32}$ assigning a numeric value to abstract locations and the concretization function $\gamma : D^\sharp \to \mathcal{P}(\mathbb{B}_{32})$ of the numeric domain. The concretization is then obtained using standard constructions:

$$\gamma_\lambda(L, o) = \{v + \lambda(L) \mid v \in \gamma(o)\}$$
$$\gamma_\lambda(\rho^\sharp) = \{\rho \mid \forall r. \rho(r) \in \gamma_\lambda(\rho^\sharp(r))\}$$
$$\gamma_\lambda(\phi^\sharp) = \{\phi \mid \forall i \in [0, f_s]. \phi(i) \in \gamma_\lambda(\phi^\sharp(i))\}$$
$$\gamma_\lambda(\langle \rho^\sharp, \phi^{\sharp(b)}, \iota \rangle) = \left\{ \langle \rho, \phi^{(\beta)}, \iota \rangle \,\middle|\, \beta \geq b \wedge \rho \in \gamma_\lambda(\rho^\sharp) \wedge \phi \in \gamma_\lambda(\phi^\sharp) \right\}$$

The mapping λ denotes a set of intra-procedural contexts such that a register r in the environment ρ_i has the value $\lambda(r)$ and the return address is constrained by the calling conventions.

$$\gamma(\lambda) = \left\{ \langle \phi_i, bp, \rho_i \rangle \,\middle|\, \begin{array}{l} \forall r, \rho(r) = \lambda(r), \\ bp = \lambda(\textsc{bp}) = \lambda(\textsc{esp}), \quad isret(\lambda(\textsc{ret}), \rho_i, \phi_i) \end{array} \right\}.$$

Finally, the whole concretization $\gamma : State^\sharp \to \mathcal{P}(Ctx^\sharp \times State^\natural)$ is defined as:

$$\gamma(s^\sharp) = \{\langle \Gamma, \blacksquare \rangle\} \cup \{\langle \Gamma, s \rangle \mid \exists \lambda, \Gamma \in \gamma(\lambda) \wedge s \in \gamma_\lambda(s^\sharp)\}.$$

4.2 Abstract Semantics

The abstract semantics takes the form of a transition system that is presented in Fig. 4. The rule AAssign abstracts the assignment to a register and consists in evaluating the expression e using the abstract domain of Section 4.1. A memory

---

[1] This exploits the property that the range of the sandbox is a power of 2.

$$\mathrm{AAssign}\ \frac{instr(\iota) = \lfloor r = e\rfloor}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho[r \mapsto [\![e]\!]_\rho], \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{AStD}\ \frac{instr(\iota) = \lfloor [e_1] = e_2\rfloor \quad [\![e_1]\!]_\rho = (\textsc{data}, o) \quad \gamma(o) \subseteq [0; d_s[}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho, \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{AStF}\ \frac{instr(\iota) = \lfloor [e_1] = e_2\rfloor \quad [\![e_1]\!]_\rho = (\textsc{bp}, o) \quad off \in \gamma(o) \quad \gamma(o) \subseteq]- \mid \phi \mid; 0]}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho, \phi[off \mapsto [\![e_2]\!]_\rho]^{(1)}, \iota^+\rangle}$$

$$\mathrm{ALdD}\ \frac{instr(\iota) = \lfloor r = [e]\rfloor \quad [\![e]\!]_\rho = (\textsc{data}, o) \quad \gamma(o) \subseteq [0; d_s[}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho[r \mapsto (\textsc{zero}, \top)], \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{ALdF}\ \frac{instr(\iota) = \lfloor r = [e]\rfloor \quad [\![e]\!]_\rho = (\textsc{bp}, o) \quad off \in \gamma(o) \quad \gamma(o) \subseteq]- \mid \phi \mid; 0]}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho[r \mapsto \phi(off)], \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{ALdS}\ \frac{instr(\iota) = \lfloor r = [e]\rfloor \quad [\![e]\!]_\rho = (\textsc{bp}, o) \quad \gamma(o) \subseteq]0; \mid \phi \mid [}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho[r \mapsto (\textsc{zero}, \top)], \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{ACall}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e\rfloor \quad \gamma([\![e]\!]_\rho) \subseteq \mathcal{F} \cup \mathcal{T} \quad \rho(\mathbf{esp}) = (\textsc{bp}, o) \\ off \in \gamma(o) \quad \mid \phi_1 \mid = -off \quad \gamma(o) \subseteq]-f_s; 0] \\ isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad \mid \phi_2 \mid = \mid \phi_2' \mid \end{array}}{\langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho', \phi_1 \cdot \phi_2'^{(1)}, \iota^+\rangle}$$

$$\mathrm{ARet}\ \frac{instr(\iota) = \lfloor \mathbf{ret}\ e\rfloor \quad preserve(\rho) \quad [\![e]\!]_\rho = (\textsc{ret}, o) \quad \{0\} = \gamma(o)}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \blacksquare}$$

$$\mathrm{ACont}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2\rfloor \quad 0 \in \gamma([\![e_1]\!]_\rho)}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho, \phi^{(\beta)}, \iota^+\rangle}$$

$$\mathrm{AJump}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2\rfloor \quad c \in \gamma([\![e_1]\!]_\rho) \quad c \neq 0 \\ [\![e_2]\!]_\rho = (\textsc{code}, o) \quad \iota_2 \in \gamma(o) + c_0 \quad \iota_2 \in Code \end{array}}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow^{\sharp} \langle \rho, \phi^{(\beta)}, \iota_2\rangle}$$

$$\mathrm{AHalt}\ \frac{instr(\iota) = \lfloor \mathbf{hlt}\rfloor}{\langle \rho, \phi^{(\beta)}, \iota\rangle \longrightarrow \blacksquare} \qquad \mathrm{ACrash}\ \frac{}{\blacksquare \longrightarrow \blacksquare}$$

Fig. 4: Abstract semantics

store is modelled by the rules AStD and AStF depending on whether the address is within the sandbox or within the current stack frame. Both rules ensure that the offset is within the bounds of the memory region. A memory load is modelled by the rules ALdD, ALdF or ALdS depending on whether the address is within the sandbox, the current stack frame or the caller stack frame. Each memory access is protected by a verification condition ensuring that the offset is within the relevant bounds. For the ALdF rule, the memory offset *off* is used to fetch the abstract value from the abstract frame ϕ. As the sandbox and the caller frame are not represented, we get the top element of the abstract domain i.e. $(\textsc{zero}, \top)$. The rule ACall models function calls. It checks whether the target of the call is a trusted ($f \in \mathcal{T}$) or untrusted function ($f \in \mathcal{F}$). For the call to proceed, the stack pointer **esp** must be within the bounds of the

current stack frame and the return address ι^+ needs to be stored according to the calling conventions ($isret(\iota^+, \rho, \phi_1)$). After the call, the resulting environment ρ' satisfies that the callee-saved registers are restored to their values in ρ ($\rho \sim \rho'$) and the suffix of the current frame ϕ'_2 is arbitrary i.e. $\phi'_2 = (\textsc{zero}, \top)^{|\phi_2|}$. The rule A$\textsc{Ret}$ ensures that the expression e evaluates to the return of the current function ($[\![e]\!]_\rho = (\textsc{ret}, o)\quad \{0\} = \gamma(o)$), and also that the callee-saved registers are restored to their initial values. For instance, for **ebx**, $preserve(\rho)$ ensures that $\rho(\mathbf{ebx}) = (EBX, o)$ with $\gamma(o) = \{0\}$. The rules A$\textsc{Cont}$ and A$\textsc{Jump}$ model control-flow transfer and check that the obtained code pointer is within the bounds of the code. The last two rules A$\textsc{Halt}$ and A$\textsc{Crash}$ model the crash state that is produced by the **hlt** instruction and is its own successor.

Like the intra-procedural semantics, the abstract semantics is safe if it is not stuck (Definition 4).

Definition 4 (Abstract progress). *The reachable intra-procedural states are safe (written A-Safe(A-Acc)) iff* $\forall f \in \mathcal{F}, \forall s \in \textit{A-Acc}(f).\exists s'.s \to s'$.

The abstract semantics embeds abstract verification conditions that are only sufficient but not necessary for the intra-procedural semantics. As a result, it only computes a safe approximation under the condition that all the reachable abstract states are safe.

Lemma 2 (Correctness of the abstract semantics).

$$\textit{A-Safe}(\textit{A-Acc}) \Rightarrow \textit{I-Acc} \subseteq \gamma(\textit{I-Acc}) \wedge \textit{I-Safe}(\textit{I-Acc})$$

The proof can be found in Appendix D. By transitivity, using Lemma 2 and Lemma 1, we get Theorem 1.

Theorem 1 (Correctness of SFI Verifier). $\textit{A-Safe}(\textit{A-Acc}) \Rightarrow \textit{Safe}(\textit{Acc})$

By definition of *Safe*, Theorem 1 means that the defensive semantics is not stuck.

5 Implementation and Experiments

We have implemented the static analysis on top of the BinCAT binary code analysis toolkit [2]. First, our SFI analyser reconstructs the structure of the binary and in particular partitions the code into separate functions and transforms the binary instructions into the REIL [4] intermediate representation. Second, each previously identified function is analysed separately, using the abstraction described in Section 4. For each function, the analysis checks that all the intra-procedural jumps stays within the current function and that the abstract semantics never blocks. The analysis also checks that all calls are towards previously identified entry points thus validating *a posteriori* that the initial partition of the code into distinct functions is indeed correct.

The analysis has been tested on three test suites: correctly sandboxed programs, incorrectly sandboxed programs, and optimised, correctly sandboxed programs. The first test suite is built by compiling programs that are part of the

CompCert test suite with a modified version of CompCert that includes sandboxing instructions. Because these binaries are correct by construction, the verifier should accept all of them. In our experiments, we have tested 10 programs, composed of 51 functions in total. 41 functions are verified in under 100ms, 9 functions are verified in less than 300ms and 1 function is verified in 3.5s. This last function occurs in sha3.c, and is responsible for the program being verified in 3.5s. This file is 200LoC long, while another file, aes.c, is 1.5KLoC long, composed of 7 functions and takes only 1s to validate. This suggests that the time complexity depends on the number of nested loops rather than on the size of the code to verify.

The second test suite for catching incorrect programs has been obtained by compiling incorrectly sandboxed programs with gcc and verifying they do not pass our verification. Each test in the suite aims at a different error: returning before the end of a function, writing above and below the frame, stack or sandbox and bypassing the guard zones. Overall, the test suite contains 9 programs and all are correctly identified as violating the sandbox property. Some of these programs can be found in Fig. 5.

```
asm("sub $5000000, %esp\n\t"
    "push $1");
```

(a) Attempt to write below the stack

```
data[-5] = 0;
```

(b) Attempt to write outside of the sandbox

```
int f(int *e) {
   int i;
   asm("push $main\n\t"
       "mov %1, %%ebp\n\t"
       "add %%ebp, (%%esp)\n\t"
       "ret"
       : "=r"(i)
       : "r"(*((int *)(sandbox(e, 4))))
       : "%ebp");
   return i+5;
}
```

(c) Attempt to return before the end

Fig. 5: Violation of sandboxing

We have also evaluated the ability of the analysis to verify programs where redundant sandboxing instructions have been optimised away. For instance, the sandboxing of consecutive accesses to an array can be factorised and implemented by a single sandboxing instruction. In addition to masking the most significant bits, this sandboxing instruction also zeroes out several least significant bits thus aligning the base address of the array. The reasoning is that if an address a of the sandbox is aligned on k bits we have that $a+i$ for $i \in [0, 2^k - 1]$ is also in the sandbox. We have sandboxed the programs manually and compiled them with gcc and verified whether they passed our verification. Our numerical domain is able to model alignment constraints and the analysis accepts programs

where consecutive writes are protected by the previous sandboxing operation. Yet, the analysis rejects programs where the sandboxing instruction is factored outside loops because the information inferred about the loop bound is currently not precise enough. More precision could be obtained by using more sophisticated numerical domains. An example program that fails our verification is given in Fig. 6.

```
char *a = (sfi + (t & 0b11111000)
for(char i=0; i<5; i++) {
  a[i] = i;
}
```

Fig. 6: Optimising array accesses in a loop

The use of alloca(size_t size) is another example of a code that respects the security property as defined by the defensive semantics, but cannot be understood by the analyser, unless more work is done by the programmer with the result of that function. Because we limit the maximum size of the stack frame, this function cannot work properly when its argument is bigger than this limit. The result is a pointer outside of the frame, and the module is rejected when a write at this address is detected.

6 Related Work

Software Fault Isolation has been proposed by Wahbe *et al.* [15] as a way to ensure that a binary code runs inside a sandbox. Native Client (NaCl) [16, 13] is a state-of-the-art implementation that was part of chromium-based browsers in order to safely run binary plugins. The NaCl binary verifier only performs a linear scan of the code. It is very fast, simple and has a small TCB. As shown by the RockSalt project [12], it is also amenable to formal verification. The NaCl verifier requires setting up trampoline code to share a runtime stack between trusted and untrusted code. Using abstract interpretation, we propose a more flexible binary verifier where redundant sandboxing operations can be optimised away and where the runtime stack is safely shared between trusted and untrusted code. Our TCB is bigger than NaCl but could be reduced using certified abstract interpretation [6].

Kroll *et al.* [8] propose to implement SFI as a CMINOR compiler pass of the CompCert [9] verified compiler. They show that proving both safety and security of the SFI pass is enough to get a secure binary. For their approach, redundant sandboxing operations can be optimised away by the compiler back-end; the runtime stack is managed by the compiler and therefore shared between trusted and untrusted code. A main difference with our work is that we explicitly state, using our defensive semantics, the security property that holds at binary level. Moreover, we propose a flexible binary verifier for this property.

The NaCl technology has recently been replaced by WebAssembly [5]. Webassembly is an intermediate language, similar to CMINOR, that is *just-in-time*

compiled into binary code. As the code may be malicious, the just-in-time compiler adds runtime checks to make sure the code runs inside a sandbox. Compared to the previous approaches, the TCB includes the just-in-time compiler and there is no independent binary verifier.

Binary Analysis frameworks e.g. BAP [3] and Angr [14] propose rich APIs to develop static analyses on top of an architecture independent intermediate language. We use BinCAT [2] based on the REIL intermediate representation [4]. Our binary verifier has the rare feature of being intra-procedural. As a result, we have adapted their interprocedural analysis engine and implemented our own analysis domains. Our analysis domain is inspired from the *a-locs* domain of Balakrishnan and Reps [1] which we adapt to a purely intra-procedural setting and where *a-locs* are also used to model the initial values of registers. There are full-fledged, whole-program, binary analysers e.g. Jackstab [7] and Bindead [10]. They both use a sophisticated combination of abstract domains. Our domains are simpler but specialised to prove the sandboxing property. Moreover, our analysis is intra-procedural and finely models the calling conventions.

7 Conclusions

We have shown that the Software Fault Isolation mechanism for safely executing untrusted binaries can be formalised as a defensive semantics of an intermediate representation of binary code. Our semantics generalises existing approaches and defines a relaxed SFI property where the runtime stack is safely shared between trusted and untrusted code. Using abstract interpretation, we derive from the defensive semantics an intra-procedural binary verifier which for each individual function can verify that memory accesses are sandboxed and that the code abides to calling conventions. The verifier is implemented and our tests show that it is able to validate programs even when compiler optimisations are enabled.

Further work will concern improving the robustness of the verifier and ensuring a degree of completeness *w.r.t.* more complex optimisations. To do so, we intend to enrich our abstract domains to cope specifically with program transformations based on code motion where sandboxing instrumentations are factored outside loops. In addition, we intend to extend the verifier to handle multi-threaded applications. We expect that the data-local, intra-procedural design of the verifier will greatly facilitate the extension to such a multi-threaded setting.

References

1. G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.
2. P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger. BinCAT: purrfecting binary static analysis. In *Symp. sur la sécurité des technologies de l'information et des communications*, 2017.
3. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011.

4. T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest'09*, 2009.

5. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proc. of the 38th Conf. on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

6. J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *Proc. of the 42Nd Symp. on Principles of Programming Languages*, pages 247–259. ACM, 2015.

7. J. Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, November 2010.

8. J. A. Kroll, G. Stewart, and A. W. Appel. Portable software fault isolation. In *Proc. of the 27th IEEE Computer Security Foundations Symp.*, pages 18–32. IEEE, 2014.

9. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

10. B. Mihaila. *Adaptable Static Analysis of Executables for proving the Absence of Vulnerabilities*. PhD thesis, Technische Universität München, 2015.

11. A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. of the Workshops on Automated Theory eXploration and on Invariant Generation*, volume 17 of *EPiC Series in Computing*, pages 55–70. EasyChair, 2012.

12. G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: Better, Faster, Stronger SFI for the x86. *SIGPLAN Not.*, 47(6):395–404, June 2012.

13. D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proc. of the 19th USENIX Conf. on Security*, pages 1–12. USENIX, 2010.

14. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symp. on Security and Privacy*, 2016.

15. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, 1993.

16. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010.

A Concrete Operational Semantics

$$\text{ISTR} \frac{instr(\iota) = \lfloor r = e \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho[r \mapsto [\![e]\!]_\rho], \mu, \iota^+ \rangle}$$

$$\text{ISTM} \frac{instr(\iota) = \lfloor [m] = e \rfloor \quad Writable([\![m]\!]_\rho)}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu[[\![m]\!]_\rho \mapsto [\![e]\!]_\rho], \iota^+ \rangle}$$

$$\text{IWCRASH} \frac{instr(\iota) = \lfloor [m] = e \rfloor \quad \neg Writable([\![m]\!]_\rho)}{\langle \rho, \mu, \iota \rangle \longrightarrow \blacksquare}$$

$$\text{ILDM} \frac{instr(\iota) = \lfloor r = [m] \rfloor \quad Readable([\![m]\!]_\rho)}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho[r \mapsto \mu([\![m]\!]_\rho)], \mu, \iota^+ \rangle}$$

$$\text{IRCRASH}\ \frac{instr(\iota) = \lfloor r = [m] \rfloor \quad \neg Readable(\llbracket m \rrbracket_\rho)}{\langle \rho, \mu, \iota \rangle \longrightarrow \blacksquare}$$

$$\text{IJCCNO}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ cond\ addr \rfloor \quad \llbracket cond \rrbracket_\rho = 0}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, \iota^+ \rangle}$$

$$\text{IJCC}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ cond\ addr \rfloor \quad \llbracket cond \rrbracket_\rho \neq 0)}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, \llbracket addr \rrbracket_\rho \rangle}$$

$$\text{ICALL}\ \frac{instr(\iota) = \lfloor \mathbf{call}\ addr \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, \llbracket addr \rrbracket_\rho \rangle}$$

$$\text{IRET}\ \frac{instr(\iota) = \lfloor \mathbf{ret}\ addr \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, \llbracket addr \rrbracket_\rho \rangle} \qquad \text{IHLT}\ \frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \blacksquare}$$

B   Intra-procedural Semantics

$$\text{FUNASSIGN}\ \frac{instr(\iota) = \lfloor r = e \rfloor}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto \llbracket e \rrbracket_\rho], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNSTD}\ \frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = d_0 + o \quad 0 \leq o < \mid \delta \mid}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNSTF}\ \frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = bp - o \quad 0 \leq o < \mid \phi \mid}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi[o \mapsto \llbracket e_2 \rrbracket_\rho]^{(1)}, \iota^+ \rangle}$$

$$\text{FUNLDD}\ \frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = d_0 + o \quad 0 \leq o < \mid \delta \mid}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto v], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNLDARG}\ \frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = (bp + f_s) - o \quad 0 \leq o < f_s}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto \phi_i(o)], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNLDFRAME}\ \frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = bp - o \quad 0 \leq o < f_s}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto \phi(o)], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNCONT}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = 0}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\text{FUNJUMP}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho \neq 0\, \llbracket e_2 \rrbracket_\rho \in Code}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi^{(\beta)}, \llbracket e_2 \rrbracket_\rho \rangle}$$

$$\text{FUNCALL}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{F} \cup \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad 0 \leq o < f_s \quad \mid \phi_1 \mid = o \\ isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle \rightarrow^\natural \langle \rho', \phi_1 \cdot \phi_2'^{(1)}, \iota^+ \rangle}$$

$$\text{FUNRET}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{ret}\ e \rfloor \quad \llbracket e \rrbracket_\rho = ret \\ isret(\iota^+, \rho_i, \phi_i) \quad \rho_i \sim \rho \end{array}}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \blacksquare}$$

$$\text{FUNHALT}\ \frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \blacksquare} \qquad \text{FUNCRASH}\ \frac{}{\Gamma \vdash \blacksquare \rightarrow^\natural \blacksquare}$$

C   Proof of Lemma 1

First we need an intermediate lemma:

Lemma 3 (Base pointer is contained).

$$\forall S = \langle\langle bp, cs, \rho_i\rangle, \langle\rho, \delta, \phi^{(\beta)}, \iota\rangle\rangle \in Acc,$$
$$\quad if\ \beta = 1$$
$$then\ s_0 - s_s + \mathrm{GZ}_\perp < bp \leq s_0 - \mathrm{GZ}_\top$$
$$else\ s_0 - s_s + \mathrm{GZ}_\perp - f_s < bp \leq s_0 - \mathrm{GZ}_\top$$

Proof. We reason by induction on $S \in Acc$. In the inital state, we have $bp = s_0 - \mathrm{GZ}_\top$ and $\beta = 0$, so the property is true since $s_s > \mathrm{GZ}_\top + \mathrm{GZ}_\perp$.

In the inductive case, we procede by case analysis on $S \Rightarrow S'$, where S verifies the property. Because the property only depends on the context and β, most cases are trivial: they preserve the context and β. In the case of the STOREFRAME rule, the context is preserved, and β is updated to 1. The property is still preserved because the property when $\beta = 0$ implies the property when $\beta = 1$.

The last case is when the extended call rule applies. In that case, $\beta(S) = 1$, $bp(S') = bp(S) - |\phi_1|$ with $|\phi_1| \leq f_s$ and $\beta(S') = 0$.

Since $s_0 - s_s + \mathrm{GZ}_\perp < bp(S) \leq s_0 - \mathrm{GZ}_\top$, $s_0 - s_s + \mathrm{GZ}_\perp - |\phi_1| < bp(S') \leq s_0 - \mathrm{GZ}_\top$, so $s_0 - s_s + \mathrm{GZ}_\perp - f_s < bp(S') \leq s_0 - \mathrm{GZ}_\top$ and the property holds.

Now we can prove the main lemma:

Proof. First, we prove that $Acc \subseteq \gamma(I\text{-}Acc)$.

Let $S \in Acc$. By induction on S, we have the following cases:

- $S = \langle\Gamma_0, \Sigma_0\rangle = \langle\langle[\langle s_0, \phi_i\rangle], s_0 - \mathrm{GZ}_\top, \rho_0\rangle, \langle\rho, \delta, \phi^{(0)}, \iota_0\rangle\rangle$ with $|\phi_i| = \mathrm{GZ}_\top$.
  By definition, $\iota_0 \in \mathcal{F}$, so we can construct $Init(\iota_0)$. By construction, $S \in \gamma(Init(\iota_0))$.
- $S = \langle\Gamma_2, \Sigma_2\rangle$ with $\langle\Gamma_1, \Sigma_1\rangle \in Acc$ and $\langle\Gamma_1, \Sigma_1\rangle \Rightarrow \langle\Gamma_2, \Sigma_2\rangle$. By induction hypothesis, we also have $S^\natural$ such that $\langle\Gamma_1, \Sigma_1\rangle \in \gamma(S^\natural)$.
  Since $I\text{-}Safe(I\text{-}Acc)$, $S^\natural \rightarrow^\natural S_2^\natural$.
  By case analysis on the rule that allows $\rightarrow^\natural$, we have:
  - *(FunAssign)*   The preconditions are the same as for (Assign), so there is S' such that $S \rightarrow^\natural S'$. Furthermore, $S' \in \gamma(s'^\natural)$.
  - *(FunStD, FunLdD, FunCont, FunIndirectJump, FunHalt, FunCrash)* Similar reasoning.
  - *(FunStF)*   Here, the preconditions are either true for (StoreFrame) or (StoreCrash) because of Lemma 3, so there is S' such that $S \rightarrow^\natural S'$, with $S' = \blacksquare$ (writing in the guard zone) or $S' = \langle\rho, \delta, \phi^{(\beta)}, \iota\rangle$ (writing in the frame). Futhermore, $S' \in \gamma(s'^\natural)$.
  - *(FunLdS)*   Here, the preconditions are either true for (LoadStack) or (LoadCrash) because of Lemma 3, so there is S' such that $S \rightarrow^\natural S'$, with $S' = \blacksquare$ (reading in the guard zone) or $S' = \langle\rho, \delta, \phi^{(\beta)}, \iota\rangle$ (reading in the stack). Futhermore, $S' \in \gamma(s'^\natural)$.

- *(FunCall)* Here the preconditions are the same as for (CallAcc) because of Lemma 3, so there is S' such that $S \Rightarrow S'$. Furthermore, $S' \in \gamma_s(Init(f)) \subseteq \gamma_s(S^\sharp)$.
- *(FunRet)* Here the preconditions are the same as for (RetAcc), so $S \Rightarrow \blacksquare$. Furthermore, $\blacksquare \in \gamma(s'^\sharp)$.

Hence our intermediate conclusion: $Acc \subseteq \gamma(I\text{-}Acc)$.

Let's now take $S \in Acc$. We use the previous conclusion to also choose $S^\sharp \in I\text{-}Acc$ such that $S \in \gamma(S^\sharp)$. Because we have $I\text{-}Safe(I\text{-}Acc)$, we can also take $S_2^\natural$ such that $S^\natural \to S_2^\natural$.

By case analysis with a similar reasoning as the previous property, we get that $S \Rightarrow S'$ with $S' \in \gamma(S_2^\natural)$.

Hence $Safe(Acc)$.

D   Proof of Lemma 2

Proof. First, we prove that $I\text{-}Acc \subseteq \gamma(A\text{-}Acc)$.

Let $S \in I\text{-}Acc$. By induction on S, we have the following cases:

- $S \in Init(f) = \langle\langle \phi_i, bp, \rho\rangle, \langle \rho, \phi^{(f)}, 0\rangle\rangle$
  We can construct $S^\sharp = \langle\langle \phi_i', bp, \rho'\rangle, \langle \rho', \phi'^{(f)}, 0\rangle \in AInit(f)$ such that $S \in \gamma(S^\sharp)$.
  $S^\sharp \in A\text{-}Acc$, so the property is true in that case.
- $S = \langle \Gamma, \Sigma_2 \rangle$ with $\langle \Gamma, \Sigma_1 \rangle \in I\text{-}Acc$ and $\Gamma \vdash \Sigma_1 \to^\natural \Sigma_2$. By induction hypothesis, we also have $S^\sharp$ such that $\langle \Gamma_1, \Sigma_1 \rangle \in \gamma(S^\sharp)$.
  Because we have $A\text{-}Safe(A\text{-}Acc)$, we also have $S_2^\sharp$ such that $S^\sharp \to S_2^\sharp$.
  By case analysis on $\to$, we can see as before that the preconditions of the abstract semantics are the same or more restrictive than those of the intra procedural semantics. It is also built in a way that $\langle \Gamma, \Sigma_2 \rangle \in \gamma(S_2^\sharp)$.

Hence our intermediate conclusion: $I\text{-}Acc \subseteq \gamma(A\text{-}Acc)$.

Let's now take $S \in I\text{-}Acc$. We use the previous conclusion to also choose $S^\sharp \in A\text{-}Acc$ such that $S \in \gamma(S^\sharp)$. Because we have $A\text{-}Safe(A\text{-}Acc)$, we can also take $S_2^\sharp$ such that $S^\sharp \to S_2^\sharp$.

By case analysis with a similar reasoning as the previous property, we get that $S \Rightarrow S'$ with $S' \in \gamma(S_2^\sharp)$.

Hence $Safe(I\text{-}Acc)$.