Safer Unsafe Code for .NET

Pietro Ferrara
École Polytechnique, France,
Università Ca' Foscari, Italy
pietro.ferrara@polytechnique.edu

Francesco Logozzo
Microsoft Research, USA
logozzo@microsoft.com

Manuel Fähndrich
Microsoft Research, USA
maf@microsoft.com

Abstract

The .NET intermediate language (MSIL) allows expressing both statically verifiable memory and type safe code (typically called managed), as well as unsafe code using direct pointer manipulations. Unsafe code can be expressed in C# by marking regions of code as *unsafe*. Writing unsafe code can be useful where the rules of managed code are too strict. The obvious drawback of unsafe code is that it opens the door to programming errors typical of C and C++, namely memory access errors such as buffer overruns. Worse, a single piece of unsafe code may corrupt memory and destabilize the entire runtime or allow attackers to compromise the security of the platform.

We present a new static analysis based on abstract interpretation to check memory safety for unsafe code in the .NET framework. The core of the analysis is a new numerical abstract domain, Strp, which is used to efficiently compute memory invariants. Strp is combined with lightweight abstract domains to raise the precision, yet achieving scalability.

We implemented this analysis in Clousot, a generic static analyzer for .NET. In combination with contracts expressed in FoxTrot, an MSIL based annotation language for .NET, our analysis provides *static* safety guarantees on memory accesses in unsafe code. We tested it on all the assemblies of the .NET framework. We compare our results with those obtained using existing domains, showing how they are either too imprecise (*e.g.*, Intervals or Octagons) or too expensive (Polyhedra) to be used in practice.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logic and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logic and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Documentation, Reliability, Verification

Keywords Abstract domains, Abstract interpretation, Bounds checking, Pointer indexing, Design by Contract, Static analysis, .NET

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

The .NET framework provides a multi-language execution environment which promotes the safe execution of code. For instance, in (safe) C# it is not possible to have un-initialized variables, unchecked out-of-bounds runtime accesses to arrays or dangling pointers. Memory safety is enforced by the type system and the runtime: it is not possible to access arbitrary memory locations. Object creation and references are allowed freely, but object life-time is managed by a garbage collector and it is not possible to get the address of an object. As a consequence, safe C# provides a safer execution environment than C or C++.

Nevertheless, there are situations where direct pointer manipulations and direct memory accesses become a necessity. This is the case when interfacing with the underlying operating system, when implementing time-critical algorithms or when accessing memory-mapped devices. For this purpose, C# provides the ability to write unsafe code (unsafe C#). In unsafe code, it is possible to declare and operate on pointers, to perform arbitrary casts, to take the address of variables or fields. C# provides syntactic sugar to denote blocks of unsafe code, which avoids the accidental use of unsafe features. Unsafe code cannot run in untrusted environments.

Most of the checks commonly enforced by the *runtime*, such as bounds checking, are not present on pointer manipulating code. As a consequence the programmer is exposed to all the vagaries of C/C++ programming, such as buffer and array overflows, reading of un-initialized memory, type safety violations, *etc.*. Those errors are difficult to detect and track down, as no runtime exception is thrown at the error source. For instance, an application cannot immediately detect that some buffer overflow compromising its data consistency has occurred. Instead, it continues its execution in a

bad state, only to fail (much) later due to a corrupted state. Tracing back the cause of such bugs to the original memory corruption is often very complicated and time consuming.

Our work appears in the context of an ongoing effort to improve the reliability of the .NET platform by systematic use of the Design by Contracts (DbC) methodology [25] supported by static checking tools. In this scenario, static checking is enabled at each build or even in an interactive development environment to catch bugs early during development.

Our Analysis

We present a sound and scalable analysis to *statically* check memory safety in unsafe code. Scalability, without giving up precision, was a main goal for the analysis. Similar work for C does not fulfill these two requirements. For instance the analysis introduced by Wagner *et al.* [34] is not precise enough to check memory accesses that involve a pointer, a base and an offset, which we found to be pervasive in `mscorlib.dll`, the main library of the .NET framework. On the other hand, the analysis of Dor *et al.* [13, 12] is precise enough to capture these relations, but it is based on the use of the Polyhedra (Poly) abstract domain [10] which is known to have severe scalability problems[1]. The work of Simon and King [31, 32] improved on that by using an abstraction of Poly, where linear inequalities were restricted to *buckets* of two variables. However, we did not find it precise enough to match the programming style adopted in the code we analyzed. Our approach differs from earlier work in that it is based on the combination of *lightweight* and *focused* abstract domains, instead of a *monolithic*, precise domain. Each abstract domain is specialized (and optimized) toward a particular program property, and their combination provides a powerful analysis without sacrificing performance.

Our analysis is based on abstract interpretation [8]. It infers and checks the memory regions accessed by read and write operations. A region of memory is denoted by a pair $\langle \mathtt{p}, \mathtt{WB(p)} \rangle$, where `p` is a pointer and `WB(p)` stands for the WritableBytes of p, i.e., the size of the region in bytes accessible from p. We only allow positive offsets off pointers, thus `WB(p)` is always non-negative.

Differently stated, the pair stands for the range of addresses $[\mathtt{p}, \mathtt{p} + \mathtt{WB(p)} - 1]$. For instance, if `x` is an `Int32` and `p` is an `Int32`$*$, then the read operation $\mathtt{x} = *(\mathtt{p} + 2)$ is safe in the region $\langle \mathtt{p}, 12 \rangle$: It reads 4 bytes (the size of an `Int32` in .NET) starting from the address $\mathtt{p} + 8$ (as `p` is a pointer to `Int32`).

We use a combination of three domains to infer bounds on memory-accessing expressions. The core is the new abstract domain of Stripes (Strp) which captures properties of the form of $\mathtt{x} - a * (\mathtt{y}[+\mathtt{z}]) \geq b$, where a and b are integer constants, `x` and `y` are variables and `z` is an optional variable. Intuitively, a stripe constraint is used to validate the *upper* bound on memory accesses. Intervals (Intv) [8] are used to validate the *lower* bound of accesses. We use (a modified version of) the Linear equalities domain (LinEq) [18] to track equalities between variables.

We implemented our analysis in `Clousot`, a generic, intraprocedural and language-agnostic static analyzer for .NET [3, 23]. It uses `FoxTrot` contracts to refine the analysis and to support assume/guarantee reasoning for method calls. `FoxTrot` allows specifying contracts in .NET without requiring any language support. Contracts are expressed directly in the language as method calls and are persisted to MSIL using the normal compilation process of the source language. We tried our analysis on all the assemblies of the .NET framework, validating on average more than 54% of unsafe memory accesses automatically in a few minutes. In practice, the false alarms that we get are due to missing contracts: the use of contracts will allow us to improve the precision. The analysis is fast enough to be used in test builds.

Our Contribution

The main contributions of the present work can be summarized as follows:

- We introduce the first static analysis to check memory safety in unsafe managed code. Our analysis handles the entire MSIL instruction set and is fully implemented in `Clousot`. It statically checks contracts, and can use them to refine the precision of the analysis, *e.g.* by exploiting preconditions. We tested it on all the assemblies of the .NET framework.
- We define the concrete and abstract semantics for an idealized MSIL-like bytecode. We prove soundness by using the abstract interpretation framework to relate the abstract semantics with the concrete semantics.
- We present a new abstract domain for the analysis of memory bounds. It is based on the co-operation of several specialized domains. We prove its soundness, and we show how it is effective in practice, by enabling a fast, yet precise analysis.
- We discuss some implementation issues necessary to avoid loss of precision, as *e.g.* the special handling that is required for the C# `fixed` statements.

2. Examples

We illustrate the analysis with some representative examples, given in increasing order of complexity. The examples are taken from, or inspired by code patterns that we found in the .NET framework assemblies.

[1] The worst case complexity of Poly is exponential. To the best of our knowledge, at the moment of writing, the most optimized implementations do not scale to more than 40 variables [1, 2]. In the analysis of .NET assemblies, we need to capture up to 965 variables.

```
static unsafe void InitToZero(int* a, uint len)
{
  Contract.Requires(
    Contract.WritableBytes(a) >= len * sizeof(int));

  for (int i = 0; i < len; i++)
  {
    *(a + i) = 0;  // (1)
  }
}
```

Figure 1. A method that zeros a region of memory. The precondition specifies that there are at least `len` $*$ sizeof(`int`) bytes allocated starting from `a`. `Clousot` propagates the precondition, and checks that the write operation at (`1`) is within bounds.

2.1 Array initialization

As a first example, consider the `InitToZero` method in Fig. 1. It initializes the memory region $[\mathtt{a}, \mathtt{a}+4*\mathtt{len}-1]$ to zero. The precondition requires that at least `len` $*$ sizeof(`int`) bytes starting from `a` are allocated. We express it using `FoxTrot` notation: contracts are specified by static method calls (*e.g.* `Contract.Requires(...)` for preconditions), and lengths of memory regions are denoted by `Contract.WritableBytes(...)`. Section 3.1 contains more information about contracts.

The write operation at (`1`) is correct provided that: (a) $\mathtt{i} \geq 0$, and that (b) $\mathtt{WB(a)} - 4 * \mathtt{i} \geq 4$. We prove (a) using the Intv abstract domain, which infers the loop invariant $\mathtt{i} \geq 0$. We prove (b) using the Strp abstract domain, which propagates the entry state $\mathtt{WB(a)} - 4 * \mathtt{len} \geq 0$ to the loop entry point, discovering the loop invariant $\mathtt{WB(a)} - 4 * (\mathtt{i} + 1) \geq 0$.

2.2 Callee checking

Methods such as `InitToZero` that use unsafe pointers are typically internal to the .NET framework and accessible only through safe wrappers such as `FastInitToZero` shown in Fig. 2. This code casts the parameter array of `int` to a pointer to `int`, and then invokes `InitToZero`. This pattern of a safe wrapper around unsafe pointer manipulating code is pervasive in the .NET framework. Using our analysis together with method pre-conditions allows us to validate that callers into the framework cannot cause unintended memory access via the internal pointer operations.

In this example, `Clousot` figures out that at line 4 of Figure 2 the invariant $\mathtt{WB(a)} = 4 * \mathtt{arr.Length}$ holds, which is enough to prove the pre-condition of `InitToZero`. In order to track affine linear equalities as above, we use the abstract domain of LinEq. The combination of Strp, Intv and LinEq allows us to precisely analyze memory accesses in unsafe code without turning to expensive (exponential) abstract domains.

```
1 static public unsafe void FastInitToZero(int [] arr)
2 {
3   fixed (int* a = arr)
4   {
5     InitToZero(a, (uint) arr.Length);
6   }
7 }
```

Figure 2. A recurrent code pattern in `mscorlib.dll`: an array is manipulated by taking a pointer to it, and the elements are accessed directly to avoid the runtime overhead of bounds checking. The `fixed` statement "pins" an object, avoiding it to be moved by the garbage collector.

2.3 Interaction with the operating system

Unsafe code is also necessary for interfacing with the underlying operating system. Consider the code in Fig. 3. `FastCopy` uses the `CopyMemory` method from the Win32 API to copy the values of the array `s` into the array `p`. `FoxTrot` allows attaching out-of-band contracts to assemblies, and in particular to annotate external calls. For the sake of presentation, we made the out-of-band contract explicit in a proxy method.

The precondition for `CopyMemory`, informally stated in the Win32 documentation, is formalized in `CopyMemoryProxy`. It requires that (a) the destination buffer is large enough to hold `szsrc` bytes; (b) the two buffers are defined at least on the memory regions accessed by `CopyMemory`.

`Clousot` can then statically check the right usage of the API. For instance, it checks that `FastCopy` satisfies the precondition, provided that the length of the destination array is not strictly smaller than the source.

Discussion: Application to security. The example shows the relevance of our analysis to enforce security. Unsafe code in the `.NET` framework is a potential security risk if it is exploitable from safe managed code. Analyses such as `Clousot` provide more confidence that the managed to unmanaged transition does not expose the framework to such attacks. The same technique could be applied at the Java to native boundary which exhibits the same problems.

2.4 Inheritance

When combined with inheritance, unsafe code can make the code *fragile* because of implicit (or informal) contracts in the application. The example in this section shows how the combination of `FoxTrot` with `Clousot` can make the existing code more robust at almost no extra-cost.

Consider the class in Fig. 4, extracted from the namespace `System.Text`, part of `mscorlib.dll`. The method `GetBytes` encodes a set of characters into a sequence of bytes. For performance reasons it uses pointers and it is declared unsafe. It can be directly invoked by clients of the library (it is a public method of a public class), or internally by the library itself.

```
[DllImport("kernel32.dll")]
unsafe static extern void
  CopyMemory(char* pdst, char* psrc, int size);

static unsafe private void
  CopyMemoryProxy(char* pdst, char* psrc,
    int szdst, int szsrc)
{
  Contract.Requires(szdst >= 0 && szsrc >= 0);
  Contract.Requires(szdst >= szsrc);
  Contract.Requires(
    Contract.WritableBytes(pdst) >= szdst*sizeof(char));
  Contract.Requires(
    Contract.WritableBytes(psrc) >= szsrc*sizeof(char));

  CopyMemory(pdst, psrc, szsrc);
}

public unsafe static void FastCopy(char[] d, char[] s)
{
  Contract.Requires(d.Length >= s.Length);

  fixed (char* pdst = d, psrc = s)
  {
    CopyMemoryProxy(pdst, psrc, d.Length, s.Length);
  }
}
```

Figure 3. An example illustrating the invocation of the Win32 API. `FoxTrot` can produce out-of-band contracts for `CopyMemory`, but we made them explicit as a proxy. `Clousot` checks that `FastCopy` respects the precondition, provided that `d.Length >= s.Length`. As a consequence, no buffer overrun occurs, making potentially dangerous code safe.

This method is inherently dangerous for two main reasons. First, the client of the library can pass wrong parameters, *e.g.* `charCount` can be larger than the memory allocated for `chars`, causing a buffer overflow. It is the responsability of the caller to keep `charCount` in sync with the region for `chars`. The .NET Base Class Library (BCL) makes sure that pointers and indexes are correct when `GetBytes` is invoked internally (*e.g.* Fig. 5). Third-party code should obey the informal documentation, but it cannot easily detect that an overrun has occured, as no exception is thrown (*e.g.*, unlike `ArrayOutOfBoundsException` for array overflows).

Second, `GetBytes` is virtual, so clients can create a subclass of `Encoding`, override `GetBytes`, and pass an instance of it to the BCL. A buggy redefinition of `GetBytes` can compromise the stability of the runtime, even if the caller has passed the correct parameters. For instance the BCL may contain some internal code that looks like the one in Fig. 5. When invoked with an instance of `Buggy` (defined in Fig. 6),

```
1  public abstract class Encoding
2  {
3    public virtual unsafe int GetBytes(char* chars,
4      int charCount, byte* bytes, int byteCount)
5    {
6      if (bytes == null || chars == null)
7        throw new Exception();
8      if (charCount < 0 || byteCount < 0)
9        throw new Exception();
10
11     char[] arrChar = new char[charCount];
12
13     for (int index = 0; index < charCount; index++)
14       // Possible buffer overrun
15       arrChar[index] = chars[index];
16
17     // ... rest of the methdod omitted ...
18   }
19   // ... rest of the class omitted ...
20 }
```

Figure 4. An example extracted from the .NET Base Class Library (BCL). The method `GetBytes` has two potential flaws: (a) A buffer overrun at line 15 if `charCount` is larger than the length of the buffer `chars` and (b) it makes the client code *fragile*, by enabling overriden methods to do whatever they want with the `chars` and the `bytes` pointers. As `GetBytes` is also called internally in the BCL, a bug in the overriden method may compromise the stability of the whole platform.

```
private unsafe void UseGetChars(Encoding e)
{
  char* chars = stackalloc char[16];
  char* myPrivateData = stackalloc char[32];
  // ... init myPrivateData ...
  byte* localBuffer = stackalloc byte[16];
  e.GetBytes(chars, 16, localBuffer, 16);
  // ...
}
```

Figure 5. An example of the use of `GetChars`. The programmer is carefull in passing the right length for the buffers, but he cannot protect himself from wrong implementations of `GetChars` which can corrupt the local state, for instance by overwriting the content of `myPrivateData`.

the first byte of `myPrivateData` is overwritten, compromising the integrity of the private state of `UseGetChars`.

We can make the code more robust by adding suitable memory safety contracts and use `Clousot` to enforce them statically. First, it is worth noting that when executed on the class in Fig. 4 as-is, `Clousot` complains about a possible overrun at line 15. Therefore we add the following contract

```
1  class Buggy : Encoding
2  {
3    override public virtual unsafe
4      int GetBytes(char* chars, int charCount,
5        byte* bytes, int byteCount)
6    {
7      for(int index = 0; index <= charCount; index++)
8      {
9        chars[index] = 'a';   // An off-by-one
10     }
11   }
12 }
```

Figure 6. A subclass of `Encoding` which has a bug that produces a buffer overrun. When an instance is passed as actual parameter for `UseGetChars`, it corrupts the local state. `Clousot` reports that `Buggy.GetBytes` violates the inherited contract.

to the method:

$$\texttt{Requires(WritableBytes(chars)} \geq \texttt{charCount} * \texttt{sizeof(char))}.$$

Under this precondition, `Clousot` automatically verifies: (a) that the body of `Encoding.GetBytes` does not cause any overrun; and (b) `UseGetChars` in Fig. 5 estabilishes the precondition for `GetBytes`. This is now checked statically and automatically, without relying on the programmer's good will to obey the documentation. Since `FoxTrot` contracts are inherited, `Clousot` points out the off-by-one bug at line 9, in Fig. 6. The programmer of `Buggy` can then correct the bug.

Discussion. Even if `Clousot` can help to make the code more robust, it cannot solve the fragility introduced by the use of *public virtual unsafe* methods. One solution is to avoid their use. Another would be to use `Clousot` during class loading to statically check whether it respects the necessary contracts or not.

3. Background

We provide some background material on `FoxTrot`, `Clousot`, and abstract interpretation.

3.1 Foxtrot

`FoxTrot` is a language independent solution for contract specifications in .NET. It does not require any source language support or compiler modification. Preconditions and postconditions are expressed by invocations of static methods (`Contract.Requires` and `Contract.Ensures`) at the start of methods. Class invariants are contained in a method with an opportune name (`ObjectInvariant`) or tagged by a special attibute (`[ObjectInvariant]`). Dummy static methods are used to express meta-variables such as *e.g.* `Contract.Old(x)` for the value in the pre-state of `x` or

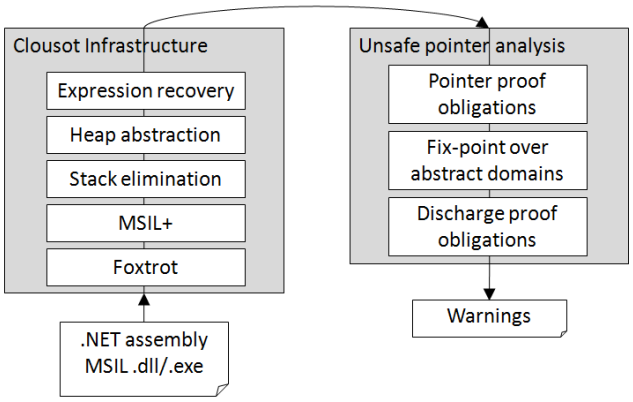


Figure 7. Clousot architecture

`Contract.WritableBytes(p)` for the length of the memory region associated with `p`. These contracts are persisted to MSIL using the standard source language compiler.

Contracts in the `FoxTrot` notation (using static method calls) can express arbitrary boolean expressions as preconditions and post-conditions. We expect the expressions to be side effect free (and only call side-effect free methods). We use a separate purity checker to optionally enforce this.

A binary rewriter tool enables dynamic checking. It extracts the specifications and instruments the binary with the appropriate runtime checks at the applicable program points, taking contract inheritance into account. Most `FoxTrot` contracts can be enforced at runtime, however contracts using `Contract.WritableBytes(...)` are a notable exception. We do not dynamically check for buffer overruns as there is no easy way to obtain the writable extend of a pointer at runtime.

For static checking, `FoxTrot` contracts are presented to `Clousot` as simple `assert` or `assume` statements. E.g., a pre-condition of a method appears as an assumption at the method entry, whereas it appears as an assertion at every call-site.

3.2 Clousot

`Clousot` is a generic, language agnostic static analyzer based on abstract interpretation for .NET. It is generic in that it presents a pluggable architecture: analyses can be easily added by providing an implementation of a suitable abstract domain interface. It is language agnostic as it analyzes MSIL. All the programming languages in .NET emit MSIL: Using the debug information we can trace back the results of the analysis to the source program.

`Clousot` has a layered structure as shown in Fig. 7. Each layer on the left presents an increasingly abstract *view* of the code. An MSIL reader sits at the lowest level, which presents a stack-based view of the code. Above that sits the `FoxTrot` extractor, which turns the dummy method calls expressing pre- and post-conditions into actual representations of these, seperating them from the method body.

ldstack.i	duplicate i-th value on evaluation stack
ldresult	load the current result value
assert	assert top of stack is true
assume	assume top of stack is true
begin_old	evaluate next instructions in method pre-state
end_old	switch back to state at matching begin_old

Table 1. MSIL+ synthetic instructions

The layer labeled MSIL+ represents an extension of MSIL with a number of synthetic instructions that allow us to express all contract code as simple stack instructions, similar to MSIL. The extensions used are listed in Table 1. Instruction `ldstack.i` is a generalization of a typical `dup` instruction that allows one to access values on the evaluation stack that are not at the top. This instruction is useful for example to access the parameters inside a pre-condition inserted at a call-site. The `ldresult` instruction is used in post-conditions to refer to the result of the method. The meaning of `assert` and `assume` is equivalent for run-time checking: they both result in failure if the condition is false. For static checking, they differ in that the checker tries to validate an `assert` condition and issues an error if it cannot be proven. However, the static checker simply adds the condition of an `assume` to its knowledge base without trying to validate it.

The next layers in the `Clousot` infrastructure (1) get rid of the stack by providing a view of the code in the 3-address form (the direct analysis of a stack-based language is hard and error-prone, [17]); (2) abstract away the heap by providing a view of the code as a *scalar* program, where aliasing has been resolved (a common approach to separate heap-analysis and value analysis, *e.g.* [5, 21]); and (3) reconstruct (most of the) expressions that have been lost during the compilation (large chunks of expressions are vital for a precise static analysis [22]).

On top of this infrastructure we build particular analyses, such as the one presented in this paper regarding unsafe memory accesses. Such analyses are built out of atomic abstract domains (*e.g.* Intv, LinEq, Pntg [23]), a set of generic domains (*e.g.* finite set of constraints), and a set of operators on abstract domains (*e.g.* the reduced cartesian product [9], the functional lifting). As a consequence `Clousot` allows building new and powerful abstract domains by refinement and composition of existing ones.

3.3 Basics of Abstract Interpretation

Abstract interpretation is a theory of approximations [8]. It formalizes the intuition that semantics are more or less precise depending on the observation level. The more precise the abstract semantics, the more precise the properties about the execution of the program it captures. A static analysis is an abstract semantics which is rough enough to be computable, and precise enough to capture the properties of interest. The design of an abstract interpreter involves: (i) the design of an abstract domain; (ii) the design of a widening operator; (iii) the design of the transfer functions.

Abstract Domains

An abstract domain $\bar{\mathsf{D}}$ is a complete lattice $\langle \mathsf{E}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, where E is the set of abstract elements, ordered according to the relation $\sqsubseteq$. The order relation $\sqsubseteq$ can be thought of as an abstraction of the logical implication [30]. The smallest abstract element is $\bot$, the largest is $\top$. The join is $\sqcup$, and the meet is $\sqcap$. With a slight abuse of notation, we will confuse an abstract domain $\bar{\mathsf{D}}$ with the set of its elements E.

The elements of an abstract domain are related to the concrete domain D (also a complete lattice), by means of a monotonic *concretization* function $\gamma \in [\bar{\mathsf{D}} \to \mathsf{D}]$. We will denote it by $\mathsf{D} \xleftarrow{\gamma} \bar{\mathsf{D}}$. If γ is a complete $\sqcap$-morphism, then there exists an *abstraction* function $\alpha \in [\mathsf{D} \to \bar{\mathsf{D}}]$, mapping concrete elements to their *best* abstract representation, [8]. In this case, we have a Galois connection between D and $\bar{\mathsf{D}}$, which we denote by $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \bar{\mathsf{D}}$. In this paper we assume the concrete domain to be the complete boolean lattice $\mathcal{P}(\Sigma)$, where Σ is the set of concrete program states.

Abstract domains can be systematically refined to augment their precision, [9]. Given two abstract domains, $\bar{\mathsf{D}}_1$ and $\bar{\mathsf{D}}_2$, their reduced cartesian product is $\bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2$, whose elements are pairs which satisfy the reduction condition:

$$\forall \langle \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \rangle \in \bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2 .\, \gamma_{\bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2}(\langle \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \rangle) \subseteq \gamma_{\bar{\mathsf{D}}_1}(\bar{\mathsf{d}}_1) \cap \gamma_{\bar{\mathsf{D}}_2}(\bar{\mathsf{d}}_2) \,.$$

Widening operator

Most of the abstract domains used in practice do not satisfy the ascending chain condition (ACC), so that the least fixpoint computation on such domains may not terminate. A widening operator is then used to extrapolate the sequence limit. Stated otherwise, it enables *dynamic* approximation. Formally, a widening operator $\triangledown \in [\bar{\mathsf{D}} \times \bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ is such that $\forall \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \in \bar{\mathsf{D}}$. $\bar{\mathsf{d}}_1 \sqsubseteq \bar{\mathsf{d}}_1 \triangledown \bar{\mathsf{d}}_2$ and $\bar{\mathsf{d}}_2 \sqsubseteq \bar{\mathsf{d}}_1 \triangledown \bar{\mathsf{d}}_2$ and for all the increasing chains $\bar{\mathsf{d}}_0 \sqsubseteq \ldots \bar{\mathsf{d}}_n \sqsubseteq \ldots$ the increasing chain defined as $\bar{\mathsf{w}}_0 = \bar{\mathsf{d}}_0, \ldots \bar{\mathsf{w}}_{i+1} = \bar{\mathsf{w}}_i \triangledown \bar{\mathsf{d}}_{i+1}$ is not strictly increasing. Then, the upward fixpoint iterations with widening will converge to a post-fixpoint [8].

Transfer functions

Given an abstract domain $\bar{\mathsf{D}}$, a transfer function $\bar{\tau} \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ is an overapproximation of the concrete semantics $\tau \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$, *i.e.* it satisfies the soundness relation $\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}$. $\tau \circ \gamma(\bar{\mathsf{d}}) \subseteq \gamma \circ \bar{\tau}(\bar{\mathsf{d}})$. Note that in general we do not require to have the most precise (complete) transfer function, just a sound (yet precise) approximation.

The Intv abstract domain

The elements of the abstract domain of intervals, Intv [8], belong to the set $\{[i, s] \mid i, s \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$. The concretization function, $\gamma_{\mathsf{Intv}} \in [\mathsf{Intv} \to \mathcal{P}(\mathbb{Z})]$ is defined as $\gamma_{\mathsf{Intv}}([i, s]) = \{z \in \mathbb{Z} \mid i \leq z \leq s\}$. The order is interval inclusion, the bottom element is the empty interval

$$\texttt{istr} ::= \begin{array}{llll} \texttt{T*\,p} & = & \texttt{stackalloc T[exp]} & \mid \\ \texttt{x} & = & *(\texttt{p} + \texttt{exp}) & \end{array} \quad \mid \quad \begin{array}{llll} \texttt{fixed(T*\,p} & = & \&\texttt{x} + \texttt{exp)\,\{\,istr\,\}} & \mid \\ *(\texttt{p} + \texttt{exp}) & = & \texttt{x} & \end{array} \quad \mid \quad \texttt{istr; istr}$$

Table 2. uMSIL: an idealized version of the MSIL instructions that are peculiar to direct memory access. T denotes a type, p a pointer, x a variable, exp a side-effects free expression.

(*i.e.*, an interval where $s < i$), the largest element is the line $[-\infty, +\infty]$. The join and the meet are respectively the union and the intersection of intervals. Intv does not satisfy the ACC, so a widening operator is required. The traditional widening on intervals preserves the bounds which are stable, [8].

Example (Widening of* Intv*) Let us consider the code in Fig. 1. The abstract values that the indexing variable i assumes during the fixpoint iterations form a strictly increasing chain:

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq ...$$

The widening keeps the stable bound (the lower bound), and extrapolates the unstable bound (the upper bound) to $+\infty$. A further iteration suffices to prove that $\texttt{i} \in [0, +\infty]$ is a fixpoint, and hence a loop invariant. □

The abstract domain of interval environments, Boxes, is the functional lifting of Intv, *i.e.*, $\mathsf{Boxes} = [\texttt{Vars} \to \mathsf{Intv}]$. The lattice operations are hence the functional extension of those defined on a single interval. The concretization of a box, $\gamma_{\mathsf{Boxes}} \in [\mathsf{Boxes} \to \mathcal{P}(\Sigma)]$ is defined as $\gamma_{\mathsf{Boxes}}(f) = \{\sigma \in \Sigma \mid \forall \texttt{x.x} \in f \implies \sigma(\texttt{x}) \in \gamma_{\mathsf{Intv}}(f(\texttt{x}))\}$. The transfer functions for the assignment and the boolean guards in the interval environment are defined as usual in interval arithmetic, [7]. The complexity of the operations and transfer functions is linear in the number of variables n: $\mathcal{O}(n)$.

In the sequel, we will not distinguish between Intv and Boxes.

The LinEq abstract domain

The elements of the LinEq abstract domain [18, 27] are sets of affine linear equalities over rationals:

$$L \in \mathcal{P}\left(\left\{\sum_i a_i * \texttt{x}_i = b \mid a_i, b \in \mathbb{Q}\right\}\right).$$

The meaning is given by the concretization $\gamma_{\mathsf{LinEq}} \in [\mathsf{LinEq} \to \mathcal{P}(\Sigma)]$:

$$\gamma_{\mathsf{LinEq}}(L) = \bigcap_{\sum_i a_i * \texttt{x}_i = b \in L} \left\{\sigma \in \Sigma \mid \sum_i a_i * \sigma(\texttt{x}_i) = b\right\},$$

therefore elements of LinEq are affine sub-spaces. The order is sub-space inclusion, the bottom is the empty space, the top is the whole space, the join is the smallest space which contains the two arguments, the meet is space intersection. LinEq satisfies the ACC condition, so that the join suffices to ensure analysis termination. The complexity of the domain operations and transfer functions is subsumed by the complexity of the Gaussian reduction that is used to provide a canonical representation for the equations. Therefore it is $\mathcal{O}(n^3)$.

4. Syntax and Concrete Semantics

We present an idealized and simplified subset of MSIL, uMSIL. We define its transition semantics. The concrete semantics is instrumented to trace the region of allocated memory associated with a pointer. We treat out-of-region memory accesses as errors.

4.1 Syntax

We focus our attention on the MSIL instructions that are particular to our unsafe analysis. Thus, we do not discuss: (a) instructions that are "*standard*" such as jumps, assignments, method invocations, *etc.* (b) issues that are orthogonal to the unsafe code analysis, such as the precise handling of tests, expressions refinement, *etc.* We refer the interested reader to [22].

The instruction set we consider, uMSIL, is shown in Tab. 2. $\texttt{T*\,p} = \texttt{stackalloc T[exp]}$ allocates exp elements of type T on the stack. In .NET, memory can be allocated in the heap in two ways : (a) use the new keyword to allocate an object or (b) directly call the underlying operating system (*e.g.* by using the HeapAlloc Win32 API). In general, the garbage collector is free to move heap allocated objects. However, the construct $\texttt{fixed(T*\,p} = \&\texttt{x} + \texttt{exp)\{istr\}}$ (a) sets a pointer p to the address $\&\texttt{x} + \texttt{exp}$; and (b) *pins* the variable p during the execution of the sequence of instructions istr, to prevent the garbage collector from moving it. The instruction $\texttt{x} = *(\texttt{p} + \texttt{exp})$ reads the value at address $\texttt{p} + \texttt{exp}$ and stores its value in x whereas $*(\texttt{p} + \texttt{exp}) = \texttt{x}$ stores at the address $\texttt{p} + \texttt{exp}$ the value of x. Finally, we have instruction sequencing.

4.2 Concrete domain

Let Vars be a set of variables, let Add be a set of addresses, $\mathbb{N}$ be the set of numerical values (note that $\texttt{Add} \subseteq \mathbb{N}$) and Ω a special state standing for a program error. For each variable $\texttt{v} \in \texttt{Vars}$ we express by $\texttt{WB(v)}$ the number of bytes on which it is defined (if it is not a pointer, the domain would not trace information about it). We let $\texttt{WB(Vars)} = \{\texttt{WB(v)} \mid \texttt{v} \in \texttt{Vars}\}$ and $\texttt{Vars}_{\texttt{WB}} = \texttt{Vars} \cup \texttt{WB(Vars)}$.

The domain of concrete execution states is

$$\mathsf{C} = ([\texttt{Vars}_{\texttt{WB}} \to \mathbb{N}] \times [\texttt{Add} \to \texttt{Byte}] \times \texttt{Add}) \cup \{\Omega\}$$

A concrete state is either: (a) a tuple consisting of an environment f mapping variables to values, a memory g mapping

$$\frac{\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g}))<0}{\mathbb{C}[\![\mathtt{T}*\mathtt{p}=\mathtt{stackalloc\ T[exp]}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\Omega}$$

$$\frac{\begin{array}{c}n=\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})),n\geq 0\\ \langle a,\mathsf{g}'\rangle=\mathsf{alloc}(\mathtt{T},n,\mathsf{g})\\ \mathsf{f}'=\mathsf{f}\quad[\mathtt{p}\mapsto a][\mathtt{WB(p)}\mapsto n*\mathsf{sizeof}(\mathtt{T})]\end{array}}{\mathbb{C}[\![\mathtt{T}*\mathtt{p}=\mathtt{stackalloc\ T[exp]}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f}',\mathsf{g}',\mathsf{t})}$$

$$\frac{\begin{array}{l}\mathtt{WB(p)}\notin\mathsf{dom}(\mathsf{f})\vee\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g}))<0\vee\\ \mathsf{f}(\mathtt{WB(p)})<\mathsf{sizeof}(\mathtt{x})+\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g}))*\mathsf{sizeof}(*\mathtt{p})\end{array}}{\mathbb{C}[\![*(\mathtt{p}+\mathtt{exp})=\mathtt{x}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\Omega}$$

$$\frac{\begin{array}{c}\mathtt{WB(p)}\in\mathsf{dom}(\mathsf{f}),n=\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})),n\geq 0\\ \mathsf{f}(\mathtt{WB(p)})\geq\mathsf{sizeof}(\mathtt{x})+n*\mathsf{sizeof}(*\mathtt{p})\\ \mathsf{g}'=\mathsf{write}(\mathsf{g},\mathsf{f}(\mathtt{p})+n*\mathsf{sizeof}(*\mathtt{p}),\mathsf{sizeof}(*\mathtt{p}),\mathsf{f}(\mathtt{x}))\end{array}}{\mathbb{C}[\![*(\mathtt{p}+\mathtt{exp})=\mathtt{x}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f},\mathsf{g}',\mathsf{t})}$$

$$\frac{\begin{array}{l}\mathtt{WB(p)}\notin\mathsf{dom}(\mathsf{f})\vee\mathsf{f}(\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})))<0\vee\\ \mathsf{f}(\mathtt{WB(p)})<\mathsf{sizeof}(\mathtt{x})+\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g}))*\mathsf{sizeof}(*\mathtt{p})\end{array}}{\mathbb{C}[\![\mathtt{x}=*(\mathtt{p}+\mathtt{exp})]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\Omega}$$

$$\frac{\begin{array}{c}\mathtt{WB(p)}\in\mathsf{dom}(\mathsf{f})\ n=\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})),n\geq 0\\ \mathsf{f}(\mathtt{WB(p)})\geq\mathsf{sizeof}(\mathtt{x})+n*\mathsf{sizeof}(*\mathtt{p})\\ v=\mathsf{read}(\mathsf{g},\mathsf{f}(\mathtt{p})+n*\mathsf{sizeof}(*\mathtt{p}),\mathsf{sizeof}(\mathtt{x}))\\ \mathsf{f}'=\mathsf{f}[\mathtt{x}\mapsto v]\end{array}}{\mathbb{C}[\![\mathtt{x}=*(\mathtt{p}+\mathtt{exp})]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f}',\mathsf{g},\mathsf{t})}$$

$$\frac{\begin{array}{rl}&\mathtt{var\ is\ a}\ T\ \mathtt{array}\\ \mathsf{f}'=\mathsf{f}\quad&[\mathtt{p}\mapsto\mathsf{f}(\mathtt{var})+(\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})))*\mathsf{sizeof}(\mathtt{T})]\\ &[\mathtt{WB(p)}\mapsto(\mathsf{eval}(\mathtt{var.length}-\mathtt{exp},(\mathsf{f},\mathsf{g})))*\mathsf{sizeof}(\mathtt{T})]\\ \mathsf{t}'=\mathsf{t}\quad&\cup\{\mathsf{f}(\mathtt{var})\}\quad\mathbb{C}[\![\mathtt{istr}]\!](\mathsf{f}',\mathsf{g},\mathsf{t}')\to(\mathsf{f}'',\mathsf{g}'',\mathsf{t}'')\end{array}}{\mathbb{C}[\![\mathtt{fixed}(\mathtt{T}*\mathtt{p}=\&\mathtt{var}+\mathtt{exp})\{\mathtt{istr}\}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f}'',\mathsf{g}'',\mathsf{t})}$$

$$\frac{\mathbb{C}[\![\mathtt{istr}_1]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\Omega}{\mathbb{C}[\![\mathtt{istr}_1;\mathtt{istr}_2]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\Omega}$$

$$\frac{\begin{array}{rl}&\mathtt{var\ is\ a\ string}\\ \mathsf{f}'=\mathsf{f}\quad&[\mathtt{p}\mapsto\mathsf{f}(\mathtt{var})+(\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g})))*2]\\ &[\mathtt{WB(p)}\mapsto(\mathsf{eval}(\mathtt{var.Length}-\mathtt{exp},(\mathsf{f},\mathsf{g})))*2]\\ \mathsf{t}'=\mathsf{t}\quad&\cup\{\mathsf{f}(\mathtt{var})\}\quad\mathbb{C}[\![\mathtt{istr}]\!](\mathsf{f}',\mathsf{g},\mathsf{t}')\to(\mathsf{f}'',\mathsf{g}'',\mathsf{t}'')\end{array}}{\mathbb{C}[\![\mathtt{fixed}(\mathtt{T}*\mathtt{p}=\&\mathtt{var}+\mathtt{exp})\{\mathtt{istr}\}]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f}'',\mathsf{g}'',\mathsf{t}\})}$$

$$\frac{\mathbb{C}[\![\mathtt{istr}_1]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to(\mathsf{f}',\mathsf{g}',\mathsf{t}')}{\mathbb{C}[\![\mathtt{istr}_1;\mathtt{istrs}_2]\!](\mathsf{f},\mathsf{g},\mathsf{t})\to\mathbb{C}[\![\mathtt{istr}_2]\!](\mathsf{f}',\mathsf{g}',\mathsf{t}')}$$

Figure 8. The concrete transition semantics. alloc, eval, `sizeof` are auxiliary functions for handling memory allocation, evaluation of pure expressions and obtraining the size of variables and types. Ω is a the error state, which blocks the computation.

addresses to bytes, and a set t of addresses of objects pinned for the garbage collector, or (b) the special value Ω denoting that an error has occurred.

4.3 Concrete transition semantics

Figure 8 formally defines the concrete transition semantics. We use some auxiliary functions: (1) $\mathsf{eval}(\mathtt{exp},(\mathsf{f},\mathsf{g}))$ evaluates a side-effect free expression exp in state (f,g); (2) $\mathsf{alloc}(\mathtt{T},n,\mathsf{g})$ returns a pair $\langle a,\mathsf{g}'\rangle$ where a is the starting address of a freshly allocated region of g containing n elements of type T, and g' is the modified memory; (3) $\mathsf{write}(\mathsf{g},a,n,v)$ returns the updated memory $\mathsf{g}[a+i\mapsto v_{[i]}\mid i\in[0,n)]$, $v_{[k]}$ denotes the *k-th* significant byte of v; (4) $\mathsf{read}(\mathsf{g},a,n)$ reads n bytes from memory g and returns them *packed* as an integer; (5) $\mathsf{sizeof}(\mathtt{T})$ and $\mathsf{sizeof}(\mathtt{x})$ return the length, expressed in bytes, respectively of an element of type T and of the variable x.

The description of the transitions in Fig. 8 follows. The semantics for `stackalloc` first evaluates exp. If it is negative, it fails. Otherwise, it allocates a new region, sets a pointer for it to p and records the length of the region, expressed in bytes, in $\mathtt{WB(p)}$.

A write operation $*(\mathtt{p}+\mathtt{exp})=\mathtt{x}$ stores a number of bytes equal to the size of the type of x in the memory location $\mathtt{p}+\mathtt{exp}*\mathsf{sizeof}(*\mathtt{p})$. If the region for p does not contain at least $\mathsf{sizeof}(\mathtt{x})+\mathtt{exp}*\mathsf{sizeof}(*\mathtt{p})$ bytes, a buffer overrun occurs, denoted by the error state Ω. The read operation is analogous.

The semantics for `fixed` is defined according to the type of `var`. In the two cases, (a) p will point to a memory address that is obtained by combining the address value $\mathsf{f}(\mathtt{var})$ and the offset $\mathtt{exp}*s$, where s is the size of the elements; (b) the address of the pinned object $\mathsf{f}(\mathtt{var})$ is added to the set of pinned objects during the execution of `st`. As for the length of the memory regions associated with p: when `var` is (a) an array, then the size of the memory region associated with p is given by the length of the array minus the offset of the first element times the size of an element; (b) a string, then p will point to an element to the internal representation of the string as an array of `char`, and the length of the memory regions is computed accordingly.

The semantics of a sequence of instructions is the compositions of the semantics, unless the result is Ω. In this case, the error state is propagated.

5. Abstract Semantics

We derive our analysis by stepwise abstraction, [7]. First, we abstract away the values read and written trough pointers and the aliasing introduced by the `fixed` instruction. Then, we

$$\frac{!(\overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0)}{\mathbb{A}[\![\mathtt{T} * \mathtt{p} = \mathtt{stackalloc}\ \mathtt{T}[\mathtt{exp}]]\!](\bar{\mathsf{f}}) \to \Omega_?} \qquad \frac{\overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0 \quad \bar{\mathsf{f}}' = \bar{\mathsf{f}} \quad [\mathtt{WB}(\mathtt{p}) \mapsto \overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) * \mathsf{sizeof}(\mathtt{T})]}{\mathbb{A}[\![\mathtt{T} * \mathtt{p} = \mathtt{stackalloc}\ \mathtt{T}[\mathtt{exp}]]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}'}$$

$$\frac{!(\overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0) \vee \mathtt{WB}(\mathtt{p}) \notin \mathsf{dom}(\bar{\mathsf{f}}) \vee !(\bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \geq \mathsf{sizeof}(\mathtt{x}) + \overline{\mathsf{eval}}(\mathtt{exp} * \mathsf{sizeof}(*\mathtt{p}), \bar{\mathsf{f}}))}{\mathbb{A}[\![*(\mathtt{p} + \mathtt{exp}) = \mathtt{x}]\!](\bar{\mathsf{f}}) \to \Omega_?} \qquad \frac{\mathtt{WB}(\mathtt{p}) \in \mathsf{dom}(\bar{\mathsf{f}}) \quad \overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0 \quad \bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \geq \mathsf{sizeof}(\mathtt{x}) + \overline{\mathsf{eval}}(\mathtt{exp} * \mathsf{sizeof}(*\mathtt{p}), \bar{\mathsf{f}})}{\mathbb{A}[\![*(\mathtt{p} + \mathtt{exp}) = \mathtt{x}]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}}$$

$$\frac{!(\overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0) \vee \mathtt{WB}(\mathtt{p}) \notin \mathsf{dom}(\bar{\mathsf{f}}) \vee !(\bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \geq \mathsf{sizeof}(\mathtt{x}) + \overline{\mathsf{eval}}(\mathtt{exp} * \mathsf{sizeof}(*\mathtt{p}), \bar{\mathsf{f}}))}{\mathbb{A}[\![\mathtt{x} = *(\mathtt{p} + \mathtt{exp})]\!](\bar{\mathsf{f}}) \to \Omega_?} \qquad \frac{\mathtt{WB}(\mathtt{p}) \in \mathsf{dom}(\bar{\mathsf{f}}) \ \overline{\mathsf{eval}}(\mathtt{exp}, \bar{\mathsf{f}}) \geq 0 \quad \bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \geq \mathsf{sizeof}(\mathtt{x}) + \overline{\mathsf{eval}}(\mathtt{exp} * \mathsf{sizeof}(*\mathtt{p}), \bar{\mathsf{f}})}{\mathbb{A}[\![\mathtt{x} = *(\mathtt{p} + \mathtt{exp})]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}}$$

$$\frac{\begin{array}{rl} & \mathtt{var\ is}\ T\ \mathtt{array} \\ \bar{\mathsf{f}}' = & \bar{\mathsf{f}}[\mathtt{WB}(\mathtt{p}) \mapsto \overline{\mathsf{eval}}(\mathtt{var.length} - \mathtt{exp}, \bar{\mathsf{f}}) * \mathsf{sizeof}(\mathtt{T})] \\ \bar{\mathsf{f}}'' = & \mathbb{A}[\![\mathtt{istr}]\!](\bar{\mathsf{f}}') \end{array}}{\mathbb{A}[\![\mathtt{fixed}(\mathtt{T} * \mathtt{p} = \&\mathtt{var} + \mathtt{exp})\{\mathtt{istr}\}]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}''} \qquad \frac{\mathbb{A}[\![\mathtt{istr}_1]\!](\bar{\mathsf{f}}) \to \Omega_?}{\mathbb{A}[\![\mathtt{istr}_1; \mathtt{istr}_2]\!](\bar{\mathsf{f}}) \to \Omega_?}$$

$$\frac{\begin{array}{rl} & \mathtt{var\ is\ a\ string} \\ \bar{\mathsf{f}}' = & \bar{\mathsf{f}}[\mathtt{WB}(\mathtt{p}) \to (\overline{\mathsf{eval}}(\mathtt{var.Length} - \mathtt{exp}, \bar{\mathsf{f}})) * 2] \\ \bar{\mathsf{f}}'' = & \mathbb{A}[\![\mathtt{istr}]\!](\bar{\mathsf{f}}') \end{array}}{\mathbb{A}[\![\mathtt{fixed}(\mathtt{T} * \mathtt{p} = \&\mathtt{var} + \mathtt{exp})\{\mathtt{istr}\}]\!](\bar{\mathsf{f}}) = \bar{\mathsf{f}}''} \qquad \frac{\mathbb{A}[\![\mathtt{istr}_1]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}'}{\mathbb{A}[\![\mathtt{istr}_1; \mathtt{istr}_2]\!](\bar{\mathsf{f}}) \to \mathbb{A}[\![\mathtt{istr}_2]\!](\bar{\mathsf{f}}')}$$

Figure 9. The abstract transition semantics for uMSIL: concrete values and pinned variables have been abstracted away. $\overline{\mathsf{eval}}$ is the lifting of eval to handle $\top$. We assume $\geq$ and $+$ to be $\top$-strict: *e.g.* $\top \geq n = n \geq \top = \top$. $!(b)$ is defined as $!(\mathtt{false}) = !(\top) = \mathtt{true}$ and $!(\mathtt{true}) = \mathtt{false}$. $\Omega_?$ is the unknown state, which causes the computation to block, signaling that an erroneus memory access has happened.

derive a generic analysis for checking buffer overruns. The analysis is parameterized by the numerical abstract domain used to evaluate region indices.

5.1 Abstracting away the values

5.1.1 The domain

We preserve just the information on memory regions. We abstract away the second and the third component of C, and we project the first component onto the memory regions, *i.e.* $\mathtt{WB}(\mathtt{Vars})$. The abstract domain is $\bar{\mathsf{C}} = ([\mathtt{WB}(\mathtt{Vars}) \to \mathbb{N} \cup \{\top\}]) \cup \{\Omega_?\}$. We add (a) $\top$ to model values that are abstracted away, (b) $\Omega_?$ to model a set of concrete states that *may* contain the error state Ω.

5.1.2 The abstract transition semantics

The abstract semantics is in Fig. 9. The abstract function $\overline{\mathsf{eval}}$ lifts its concrete counterpart to handle $\top$. $\top$ values occur for instance when exp contains a variable x whose value is read through a pointer and we do not trace the value for x. $\overline{\mathsf{eval}}$ simply propagates $\top$ through all strict operator positions, e.g., $\overline{\mathsf{eval}}(5 + \top, f) = \overline{\mathsf{eval}}(\top, f) = \top$.

The semantics is a little bit more than the projection of the concrete semantics on its first component: if $\overline{\mathsf{eval}}(\mathtt{exp}, f) = \top$, then we cannot decide if $\mathtt{exp} \geq 0$ and hence if a buffer overrun has occured. In this case, we force the transition to the $\Omega_?$ state, which means that a buffer overrun *may* occur.

For the fixed instruction, we abstract away (a) the fact that the object is pinned: in our abstract semantics we do not need to model the garbage collector; (b) the aliasing between p and $\&\mathtt{var} + \mathtt{exp}$: we are interested just in checking that memory accesses are valid.

5.1.3 Abstraction and concretization function

The concretization function returns the set of all the concrete states such that the first component is *compatible* with one of the abstract states. If the abstract state contains the unknown state $\Omega_?$, then all the concrete states are returned, included the error state Ω. As a consequence, in order to show that a program has no memory access violations, it suffices to prove that its abstract semantics in Fig. 9 never reduces to $\Omega_?$.

The next two theorems guarantee the soundness of the approach. The first states that the abstract elements are a correct approximation of the abstract ones. The second one states that no concrete behavior is forgotten in the abstract semantics.

Theorem: Soundness of the abstraction. Let $\gamma \in [\mathcal{P}(\bar{\mathsf{C}}) \to \mathcal{P}(\mathsf{C})]$ be the concretization function defined as

$$\begin{array}{rl} \gamma(\bar{\mathsf{F}}) = & \bigcap_{\bar{\mathsf{f}} \in \bar{\mathsf{F}}} \{(\mathsf{f}, \mathsf{g}, \mathsf{t}) \mid \forall \mathtt{WB}(\mathtt{p}) \in \mathsf{dom}(\bar{\mathsf{f}}).\ \bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \neq \top \\ & \qquad \Longrightarrow \mathsf{f}(\mathtt{WB}(\mathtt{p})) = \bar{\mathsf{f}}(\mathtt{WB}(\mathtt{p})) \wedge \mathtt{p} \in \mathsf{dom}(\mathsf{f})\} \\ & \cup \{(\mathsf{f}, \mathsf{g}, \mathsf{t}) \mid \Omega_? \in \bar{\mathsf{F}}\}. \end{array}$$

Then γ is a complete $\cap$-morphism, so that it exists an abstraction function $\alpha \in [\mathcal{P}(\mathsf{C}) \to \mathcal{P}(\bar{\mathsf{C}})]$ such that $\mathcal{P}(\mathsf{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\bar{\mathsf{C}})$. □

$$\frac{\mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \top}{\mathbb{F}[\![\mathtt{T} * \mathtt{p} = \mathtt{stackalloc\ T[exp]}]\!](\bar{\mathtt{f}}) \to \Omega_?} \qquad \frac{\mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \mathtt{true} \quad \bar{\mathtt{s}}' = \mathsf{assign}(\mathtt{WB(p)}, \mathtt{size} * \mathtt{sizeof(T)}, \bar{\mathtt{s}})}{\mathbb{F}[\![\mathtt{T} * \mathtt{p} = \mathtt{stackalloc\ T[exp]}]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}'}$$

$$\frac{\begin{array}{l}\mathsf{check}(\mathtt{WB(p)} \geq \mathtt{sizeof(x)} + \mathtt{exp} * \mathtt{sizeof}(*\mathtt{p}), \bar{\mathtt{s}}) = \top \\ \vee \mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \top\end{array}}{\mathbb{F}[\![*(\mathtt{p} + \mathtt{exp}) = \mathtt{x}]\!](\bar{\mathtt{s}}) \to \Omega_?} \qquad \frac{\begin{array}{l}\mathsf{check}(\mathtt{WB(p)} \geq \mathtt{sizeof(x)} + \mathtt{exp} * \mathtt{sizeof}(*\mathtt{p}), \bar{\mathtt{s}}) = \mathtt{true} \\ \mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \mathtt{true}\end{array}}{\mathbb{F}[\![*(\mathtt{p} + \mathtt{exp}) = \mathtt{x}]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}}$$

$$\frac{\begin{array}{l}\mathsf{check}(\mathtt{WB(p)} \geq \mathtt{sizeof(x)} + \mathtt{exp} * \mathtt{sizeof}(*\mathtt{p}), \bar{\mathtt{s}}) = \top \\ \vee \mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \top\end{array}}{\mathbb{F}[\![\mathtt{x} = *(\mathtt{p} + \mathtt{exp})]\!](s) \to \Omega_?} \qquad \frac{\begin{array}{l}\mathsf{check}(\mathtt{WB(p)} \geq \mathtt{sizeof(x)} + \mathtt{exp} * \mathtt{sizeof}(*\mathtt{p}), \bar{\mathtt{s}}) = \mathtt{true} \\ \mathsf{check}(\mathtt{exp} \geq 0, \bar{\mathtt{s}}) = \mathtt{true}\end{array}}{\mathbb{F}[\![\mathtt{x} = *(\mathtt{p} + \mathtt{exp})]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}}$$

$$\frac{\begin{array}{c}\mathtt{var\ is\ a}\ T\ \mathtt{array} \\ \bar{\mathtt{s}}' = \mathsf{assign}(\mathtt{WB(p)}, (\mathtt{var.length} - \mathtt{exp}) * \mathtt{sizeof(T)}, \bar{\mathtt{s}}) \\ \mathbb{F}[\![\mathtt{istr}]\!](\bar{\mathtt{s}}') \to \bar{\mathtt{s}}''\end{array}}{\mathbb{F}[\![\mathtt{fixed(T} * \mathtt{p} = \&\mathtt{var} + \mathtt{exp)\{istr\}}]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}''} \qquad \frac{\mathbb{F}[\![\mathtt{istr}_1]\!](\bar{\mathtt{s}}) \to \Omega_?}{\mathbb{F}[\![\mathtt{istr}_1; \mathtt{istr}_2]\!](\bar{\mathtt{s}}) \to \Omega_?}$$

$$\frac{\begin{array}{c}\mathtt{var\ is\ a\ string} \\ \bar{\mathtt{s}}' = \mathsf{assign}(\mathtt{WB(p)}, (\mathtt{var.length} - \mathtt{exp}) * 2, \bar{\mathtt{s}}) \\ \mathbb{F}[\![\mathtt{istr}]\!](\bar{\mathtt{s}}') \to \bar{\mathtt{s}}''\end{array}}{\mathbb{F}[\![\mathtt{fixed(T} * \mathtt{p} = \&\mathtt{var} + \mathtt{exp)\{istr\}}]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}''} \qquad \frac{\mathbb{F}[\![\mathtt{istr}_1]\!](\bar{\mathtt{s}}) \to \bar{\mathtt{s}}'}{\mathbb{F}[\![\mathtt{istr}_1; \mathtt{istr}_2]\!](\bar{\mathtt{s}}) \to \mathbb{F}[\![\mathtt{istr}_2]\!](\bar{\mathtt{s}}')}$$

Figure 10. The generic abstract semantics for memory access validity checking. It is parameterized by a numerical abstract domain endowed with two primitives: assign and check.

Theorem: Soundness of the abstract semantics. Let $\bar{\mathsf{f}} \in \bar{\mathsf{C}}$, $(\mathsf{f}, \mathsf{g}, \mathsf{t}) \in \gamma(\bar{\mathsf{f}})$ and $\mathtt{ist} \in \mathtt{uMSIL}$. If $\mathbb{A}[\![\mathtt{ist}]\!](\bar{\mathsf{f}}) \to \bar{\mathsf{f}}'$ and $\mathbb{C}[\![\mathtt{ist}]\!](\mathsf{f}, \mathsf{g}, \mathsf{t}) \to (\mathsf{f}', \mathsf{g}', \mathsf{t}')$, then $(\mathsf{f}', \mathsf{g}', \mathsf{t}') \in \gamma(\bar{\mathsf{f}}')$. □

5.2 Generic memory access analysis

If we extend uMSIL with (conditional) jumps, *e.g.* to enable loops, then the abstract semantics in Fig. 9 will no longer be computable. In particular, the expressions used for memory accesses may evaluate to infinitely many values. As a consequence, in order to cope with a more realistic scenario, we need to perform a further abstraction, to capture the values of index expressions.

We assume a numerical domain $\bar{\mathsf{N}}$ which correctly approximates $\mathcal{P}(\bar{\mathsf{C}})$ ($\langle \mathcal{P}(\bar{\mathsf{C}}), \subseteq \rangle \xleftarrow{\gamma_{\bar{\mathsf{N}}}} \langle \bar{\mathsf{N}}, \sqsubseteq \rangle$) and with two primitives: (a) $\mathsf{assign}(\mathtt{x}, \mathtt{exp}, \mathsf{s}) \in \bar{\mathsf{N}}$ which is (an over-approximation of) the assignment $\mathtt{x} := \mathtt{exp}$ in the abstract state s ($\in \bar{\mathsf{N}}$); (b) $\mathsf{check}(\mathtt{exp}, \mathsf{s}) \in \{\mathtt{true}, \top\}$ which checks whether, in the abstract state s ($\in \bar{\mathsf{N}}$), the expression exp holds (true) or it cannot be decided ($\top$).

The generic abstract semantics for checking memory safety, parameterized by $\bar{\mathsf{N}}$ is reported in Fig. 10.

6. The "right" numerical abstract domain

The generic abstract semantics in Fig. 10 can be instantiated with any numerical abstract domain containing the primitives assign and check. As a consequence the problem of checking the validity of memory accesses boils down to the problem of chosing the *right* abstract domain.

Existing numerical domains can be classified according to their precision/cost ratio. The *ideal* of a static analysis is to use the *least* expensive domain which is precise *enough* to prove the property of interests.

Let us consider the set of points $\mathcal{A}$ of Fig. 11(a) corresponding to all the possible values that $\mathtt{WB(p)}$ and index assume at some memory access $\mathtt{c} = *(\mathtt{p} + \mathtt{index})$. Geometrically, the memory access is safe as all the concrete values are included in the upper-right quadrant delimeted by the lines $\mathtt{index} = 0$ and $\mathtt{WB(p)} = \mathtt{base} + \mathtt{index} * \mathtt{sizeof}(*\mathtt{p})$. Proving it using an abstract domain $\bar{\mathsf{A}}$ requires inferring an abstract element $\bar{\mathsf{a}} \in \bar{\mathsf{A}}$ such that $\mathcal{A} \subseteq \gamma(\bar{\mathsf{a}}) \subseteq \mathcal{R}$.

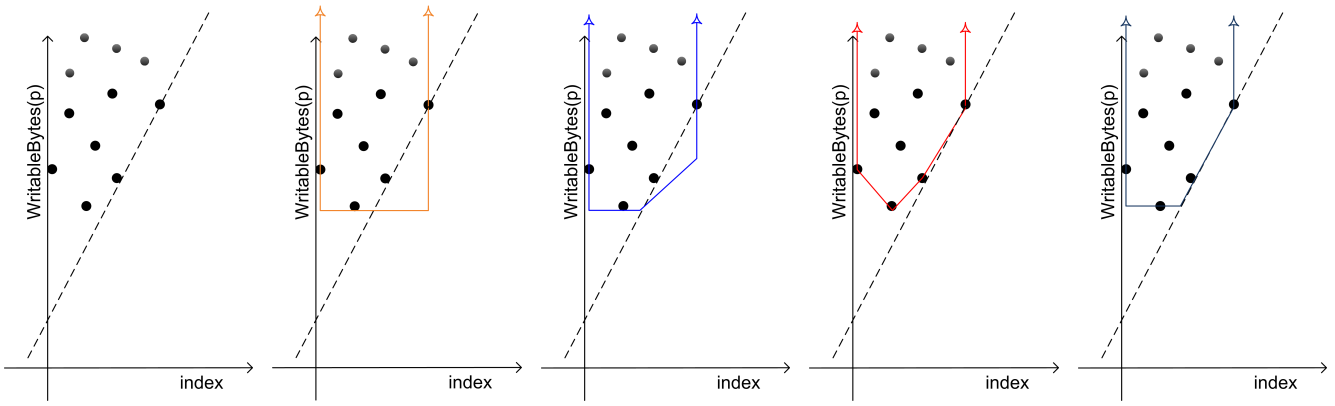
Fig. 11(b) shows that Intv alone is not precise enough for our purposes: the best approximation for $\mathcal{A}$ with Intv is not completely included in $\mathcal{R}$. Intuitively, this is because Intv does not keep relational information, *e.g.*, any relation between $\mathtt{WB(p)}$ and index is abstracted away.

Weakly-relational numerical abstract domains such as Octagons [26] or Pentagons [23] have been introduced as lightweight solutions for *array* bounds checking[2]. Fig. 11(c) shows that Octagons are more precise than Intv, but they are still not precise enough to validate memory accesses due to the multiplicative factor $\mathtt{sizeof}(*\mathtt{p})$ which makes the slopes in Fig. 11 possibly non-45°.

Fig. 11(d) shows that the convex hull $\mathsf{CH}(\mathcal{A})$ of the points $\mathcal{A}$ is included in $\mathcal{R}$. The geometrical interpretation of the elements of the abstract domain of Polyhedra (Poly) [10] is exactly their convex hull[3]. The main drawback of using Poly is its worst case cost, which for the most common

[2] Octagons capture relations in the form $\pm\mathtt{x} \pm \mathtt{y} \leq a$, and Pentagons in the form of $a \leq \mathtt{x} \leq b \ \wedge \mathtt{x} < \mathtt{y}$.

[3] Polyhedra capture arbitrary linear inequalities among variables: $\Sigma_i a_i \cdot \mathtt{x_i} \leq b$.



(a) Concrete points (b) Intervals, $\mathcal{O}(n)$ (c) Octagons, $\mathcal{O}(n^3)$ (d) Polyhedra, $\mathcal{O}(2^n)$ in practice (e) Stripes⊗Intervals⊗LinEq, $\mathcal{O}(n^3)$, $\mathcal{O}(n)$ in practice

Figure 11. The concrete points, and some approximations depending on the numerical abstract domain. Intervals and Octagons are not precise enough to prove the property. Polyhedra are precise, but also very expensive: they have an exponential complexity which shows up in practice. The reduced product of Stripes⊗Intervals⊗LinEq represents a good trade off between precision and cost: the theoretical complexity is cubic, but in practice we experienced a linear behavior.

operations is exponential in time and space (and this is a lower-bound [19]). In Sect. 9 we will provide experimental evidence that that the worst case is attained in practice.

In `Clousot` we are interested in the scalability of the analyses. Therefore, we rejected the use of general purpose, precise but very expensive abstract domains such as Poly. Instead, we have chosen a different path, which consists in (a) designing abstract domains focused on a particular property; and (b) combining domains using well-known techniques such as the reduced product. For our analysis, we designed a new numerical abstract domain, Strp, and we combined it with Intv and LinEq to achieve precision without giving up on performance. Fig. 11(e) illustrates the (best) approximation for $\mathcal{A}$ within the abstract domain Strp $\otimes$ Intv $\otimes$ LinEq, which is included in $\mathcal{R}$.

In the next sections we present the details of Strp, its reduction with Intv and LinEq, and the results of our practical experiments.

7. The Stripes abstract domain

We introduce a novel, weakly-relational domain Stripes, Strp, focused on the inference and checking of (upper bounds on) memory accesses that use a base, an index, and a multiplicative factor. We define the order, the join, the meet and the widening operators.

7.1 Constraints

As a first approximation, Strp captures constraints of the form $\texttt{WB(p)} - \mathsf{sizeof(T)} * (\texttt{count}[+\texttt{base}]) > k$ where `WB(p)`, `count`, and optionally `base` are variables, `T` is a type, and k is an integer constant. The intuition behind it is that the pointer `p` is defined at least on `count`$[+\texttt{base}]$ elements of its type, and on k additional bytes.

In practice, these constraints are used in a more generic way: the first element may be any variable (and not only the writable bytes of a pointer) and the $\mathsf{sizeof}(\texttt{T})$ may be any numerical value (and not only the size of the type of the pointer target). Then the constraints captured by the Stripe domain are $\texttt{z} - k_1 * (\texttt{x}[+\texttt{y}]) > k_2$.

7.2 Abstract domain structure

Abstract elements

We represent Strp elements as maps from variables to constraints. We have chosen maps as they allow efficient manipulation of *directional* constraints:

$$\mathsf{Strp} = [\texttt{Vars}_{\texttt{WB}} \to \mathcal{P}((\texttt{Vars}_{\texttt{WB}} \times (\texttt{Vars}_{\texttt{WB}} \cup \bot) \times \mathbb{N} \times \mathbb{N}))].$$

Intuitively, the domain of the map contains the variable `z`, the first and second component of the 4-tuple represent the two variables `x` and `y` ($\bot$ if it is not present), the third component is k_1 and the last one is k_2.

Example (Representation of stripes constraints) The two constraints $\texttt{z} - 4 * \texttt{y} > 0$ and $\texttt{z} - 2 * (\texttt{x} + \texttt{u}) \geq 5$ are represented in Strp by the map $[\texttt{z} \mapsto \{(\texttt{y}, \bot, 4, 0), (\texttt{x}, \texttt{u}, 2, 4)\}]$. □

Order

An abstract state $\bar{\mathsf{s}}_1$ in Strp is more precise than $\bar{\mathsf{s}}_2$ iff for each constraint in $\bar{\mathsf{s}}_2$, $\bar{\mathsf{s}}_1$ contains a constraint such that (a) the three variables and the integer constant k_1 are the same; and (b) k_2 is less or equal than the k_2 of $\bar{\mathsf{s}}_2$ since if $\texttt{x} > \texttt{y}$ and $\texttt{y} > \texttt{z}$ then $\texttt{x} > \texttt{z}$ by transitivity of $>$. Formally:

$$\begin{aligned}\bar{\mathsf{s}}_1 \sqsubseteq \bar{\mathsf{s}}_2 \iff \forall \texttt{z} \in \mathsf{dom}(\bar{\mathsf{s}}_2), \forall (\texttt{y}, \texttt{x}, k_1, k_2^2) \in \bar{\mathsf{s}}_2(\texttt{z}).\\ \texttt{z} \in \mathsf{dom}(\bar{\mathsf{s}}_1) \wedge\\ \exists (\texttt{y}, \texttt{x}, k_1, k_2^1) \in \bar{\mathsf{s}}_1(\texttt{z}).k_2^2 \leq k_2^1\end{aligned}$$

Top and Bottom

The largest element of Strp is a map with no information: $\lambda \mathtt{z}.\ \emptyset$. An abstract state $\bar{\mathtt{s}}$ is bottom iff it contains a contradiction: *e.g.* $[\mathtt{z} \mapsto \{(\mathtt{y}, \bot, 1, 0)\}, \mathtt{y} \mapsto \{\mathtt{z}, \bot, 1, 0)]$.

Join

The upper bound operator (a) keeps the constraints that are defined in both operands; (b) takes the smallest lower bound k_2 if it is different in the two constraints since if $\mathtt{exp} > a$, $\mathtt{exp} > b$ and $a \geq b$ then $\mathtt{exp} > b$ is an upper bound for both constraints. Formally:

$$\bar{\mathtt{s}}_1 \sqcup \bar{\mathtt{s}}_2 = \lambda \mathtt{z}.\{(\mathtt{y}, \mathtt{x}, k_1, k_2) \mid (\mathtt{y}, \mathtt{x}, k_1, k_2^1) \in \bar{\mathtt{s}}_1(\mathtt{z}), (\mathtt{y}, \mathtt{x}, k_1, k_2^2) \in \bar{\mathtt{s}}_2(\mathtt{z}), k_2 = \min(k_2^1, k_2^2)\}.$$

Meet

The lower bound operator traces the constraints of both operands. If both contain a constraint with the same variables $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$, and the same integer value k_1, the operator keeps the largest integer value for the numerical lower bound.

$$\bar{\mathtt{s}}_1 \sqcap \bar{\mathtt{s}}_2 = \lambda \mathtt{z}. \left\{ \begin{array}{r} (\mathtt{y}, \mathtt{x}, k_1, k_2) \mid (\mathtt{y}, \mathtt{x}, k_1, k_2^1) \in \bar{\mathtt{s}}_1(\mathtt{z}), \\ (\mathtt{y}, \mathtt{x}, k_1, k_2^2) \in \bar{\mathtt{s}}_2(\mathtt{z}), \\ k_2 = \max(k_2^1, k_2^2) \end{array} \right\}$$
$$\cup$$
$$\lambda \mathtt{z}. \left\{ \begin{array}{r} (\mathtt{y}, \mathtt{x}, k_1, k_2) \mid ((\mathtt{y}, \mathtt{x}, k_1, k_2) \in \bar{\mathtt{s}}_1(\mathtt{z}) \wedge \\ (\mathtt{y}, \mathtt{x}, k_1, \_) \notin \bar{\mathtt{s}}_2(\mathtt{z})) \\ \vee ((\mathtt{y}, \mathtt{x}, k_1, k_2) \in \bar{\mathtt{s}}_2(\mathtt{z}) \wedge \\ (\mathtt{y}, \mathtt{x}, k_1, \_) \notin \bar{\mathtt{s}}_1(\mathtt{z})) \end{array} \right\}$$

Widening

Strp does not satisfy the ACC condition. As a consequence, we need to define a widening operator to ensure convergence. Our widening simply drops the constraints that are not stable between two iterations:

$$\bar{\mathtt{s}}_1 \triangledown \bar{\mathtt{s}}_2 = \lambda \mathtt{z}.\bar{\mathtt{s}}_1(\mathtt{z}) \cap \bar{\mathtt{s}}_2(\mathtt{z}).$$

Concretization

The concretization function $\gamma_{\mathsf{Strp}} \in [\mathsf{Strp} \to \mathcal{P}(\bar{\mathsf{C}})]$ returns all the possible states that satisfy the constraints represented by the abstract state:

$$\gamma_{\mathsf{Strp}}(\bar{\mathtt{s}}) = \{\bar{\mathtt{f}} \mid \forall \mathtt{z} \in \mathsf{dom}(\bar{\mathtt{s}}) \forall (\mathtt{y}, \mathtt{x}, k_1, k_2) \in \bar{\mathtt{s}}(\mathtt{z}). \bar{\mathtt{f}}(\mathtt{z}) - k_1 * (\bar{\mathtt{f}}(\mathtt{y}) + \bar{\mathtt{f}}(\mathtt{x})) > k_2\}.$$

It is immediate to see that γ_{Strp} is monotonic, and furthermore that it is a complete $\cap$-morphism. Therefore, as the composition of monotonic functions is monotonic, the following theorem stating that Strp is a sound approximation holds:

Theorem (Abstraction) γ_{Strp} as defined above is a complete $\cap$-morphism. Therefore, it exists an α_{Strp} such that $\langle \mathcal{P}(\bar{\mathsf{C}}), \subseteq \rangle \underset{\alpha_{\mathsf{Strp}}}{\overset{\gamma_{\mathsf{Strp}}}{\leftrightarrows}} \langle \mathsf{Strp}, \sqsubseteq \rangle$. As a consequence, $\langle \mathcal{P}(\mathsf{C}), \subseteq \rangle \underset{\alpha_{\mathsf{Strp}} \circ \alpha}{\overset{\gamma \circ \gamma_{\mathsf{Strp}}}{\leftrightarrows}} \langle \mathsf{Strp}, \sqsubseteq \rangle$. □

7.3 Refinement of the abstract state

A state of the Stripe domain may be internally refined, by carefully propagating information between constraints.

Example (Refinement of constraints) Consider the two stripes constraints $\mathtt{x} - 2 * (\mathtt{y} + \mathtt{u}) > 4$ and $\mathtt{y} - \mathtt{z} > 0$. From the first constraint we derive:

$$\mathtt{x} - 2 * (\mathtt{y} + \mathtt{u}) > 4 \iff \mathtt{x} - 2 * \mathtt{u} - 4 > 2 * \mathtt{y} \iff \mathtt{x}/2 - \mathtt{u} - 2 > \mathtt{y}.$$

From the second constraint we derive that $\mathtt{y} > \mathtt{z} \iff \mathtt{y} \geq \mathtt{z} + 1$. Combining the two, we derive a new stripe constraint: $\mathtt{x}/2 - \mathtt{u} - 2 > \mathtt{z} + 1 \iff \mathtt{x} - 2 * (\mathtt{u} + \mathtt{z}) > 6$. □

The above example can be easily generalized:

Lemma (Saturation) If an abstract state contains the two constraints

$$\mathtt{x} - k_1 * (\mathtt{y}[+\mathtt{u}]) > k_2$$
$$\mathtt{y} - 1 * \mathtt{z} > k_3$$

then we can infer the constraint $\mathtt{x} - k_1 * (\mathtt{z}[+\mathtt{u}]) > k_2 + k_1 * (k_3 + 1)$. □

The refinement enabled by the lemma above is important in practice. It allows adding new constraints to the abstract state, without requiring an expensive closure to propagate the information. Of course, Lemma 7.3 does not guarantee the completeness of the saturation, but it is sufficent for our purposes, as illustrated by the next example.

Example (Saturation) Let us consider the example in Fig. 1. Inside the loop, we have the abstract state $\bar{\mathtt{s}} = \{\mathtt{WB(a)} - 4 * \mathtt{len} > -1, \mathtt{len} - \mathtt{i} > 0\}$ [4]. We have to check whether $\mathtt{WB(a)} \geq 4 * \mathtt{i} + 4$. We cannot do it directly by inspecting $\bar{\mathtt{s}}$ as there is no direct relation between $\mathtt{WB(a)}$ and $\mathtt{i}$. Applying the refinement of Lemma 7.3, we infer the constraint $\mathtt{WB(a)} - 4 * \mathtt{i} > 3$ which suffices to validate the access: $\mathtt{WB(a)} - 4 * \mathtt{i} > 3 \iff \mathtt{WB(a)} > 4 * \mathtt{i} + 3 \iff \mathtt{WB(a)} \geq 4 * \mathtt{i} + 4$. □

In our implementation we perform this refinement only on-demand when we need to check the proof obligations.

7.4 Transfer functions

Assignment

When an expression is assigned to a variable, we first drop all the constants that are defined on the assigned variable, and then we add some inferred constraints. Formally:

$$\mathsf{assign}(\mathtt{x}, \mathtt{exp}, \bar{\mathtt{s}}) = \mathrm{let}\ \bar{\mathtt{s}}' = \mathsf{drop}(\mathtt{x}, \bar{\mathtt{s}}) \ \mathrm{in}\ \bar{\mathtt{s}}' \cup \mathcal{C}(\mathtt{x}, \mathtt{exp}, \bar{\mathtt{s}}').$$

where

$$\mathsf{drop}(\mathtt{x}, \bar{\mathtt{s}}) = \lambda \mathtt{y}.\{(\mathtt{z}, \mathtt{u}, k_1, k_2) \mid \mathtt{y} \neq \mathtt{x}, (\mathtt{z}, \mathtt{u}, k_1, k_2) \in \bar{\mathtt{s}}(\mathtt{y}) \implies \mathtt{z} \neq \mathtt{x} \wedge \mathtt{u} \neq \mathtt{x}\};$$

[4] To simplify the reading, we present a stripe abstract state as a set of constraints.

and $\mathcal{C}$ infers new constraints from an assignment and an abstract state. Few representative cases for $\mathcal{C}$ follow. In our implementation we consider a richer structure of expressions and cases.

$$\begin{aligned}\mathcal{C}(\mathtt{x},\mathtt{y},\bar{\mathtt{s}}) =& [\mathtt{x} \mapsto \bar{\mathtt{s}}(\mathtt{y})] \cup \\ & [\mathtt{v}_1 \mapsto \{(\mathtt{x},\mathtt{v}_2,k_1,k_2) \mid (\mathtt{y},\mathtt{v}_2,k_1,k_2) \in \bar{\mathtt{s}}(\mathtt{v}_1)\}] \\ \mathcal{C}(\mathtt{x},\mathtt{u}+\mathtt{v},\bar{\mathtt{s}}) =& \\ & [\mathtt{v}_1 \mapsto \{(\mathtt{u},\mathtt{w},k_1,k_2) \mid (\mathtt{x},\bot,k_1,k_2) \in \bar{\mathtt{s}}(\mathtt{v}_1)\}] \\ & \ldots\end{aligned}$$

Abstract checking

To check a boolean expression, we first try to normalize it into a form like $\mathtt{x} - k_1 * (\mathtt{y}[+\mathtt{z}]) > k_2$, and then we check if the abstract state contains a constraint which implies it. Formally:

$$\begin{aligned}&\mathsf{check}(\mathtt{exp},\bar{\mathtt{s}}) = \\ &\mathrm{let}(\mathtt{x} - k_1 * (\mathtt{y}+\mathtt{z}) > k_2^1, b) = \mathsf{normalize}(\mathtt{exp}) \\ &\mathrm{in} \\ &\quad \text{if } (b \wedge \exists(\mathtt{y},\mathtt{z},k_1,k_2^2) \in \bar{\mathtt{s}}(\mathtt{x}).k_2^1 \leq k_2^2) \text{ then } \mathtt{true} \text{ else } \top\end{aligned}$$

We skip the details of normalize. Roughly, it applies basic arithmetic identities to rewrite the expression. If it fails to put the expression into a stripe constraint form, it returns a boolean value signaling the failure.

8. Refined Abstract Semantics

We refine the information captured by the Strp domain with Intv and the LinEq domain. Intv is needed to check lower bounds of accesses. LinEq is needed to track linear equalities, and in particular to handle the compilation schema for `fixed` in C#.

8.1 Checking lower bounds of accesses

Strp allows representing just partial numerical bounds on variables. In fact, when $k_1 = 0$, a stripe constraint boils down to a numerical lower bound: $\mathtt{z} > k_2$. Nevertheless, in general we need to track numerical upper bounds on variables: Those may appear in expressions that must be evaluated to check under-flow accesses. We use Intv to track the numerical bounds on variables.

Example (Need for numerical bounds) Let us consider the following code snippet ("..." denotes an arbitrary boolean expression):

```
int *p;
...
// suppose that WB(p) = 12, a = 5
if (...) {
  b = 3;
}
else {
  b = 4;
}
*(p + (a−b)) = 0; // (*)
```

If we track just lower bounds, at $(*)$ we have $\mathtt{a} > 4, \mathtt{b} > 2$, so that we cannot prove the memory access correct. If we track both numerical bounds, at $(*)$ we have that $\mathtt{a} = 5, \mathtt{b} \in [3,4]$, so that $\mathtt{b} - \mathtt{a} \in [1,2]$ which suffices to prove the access correct. □

The numerical abstract domain for the analysis is the product domain Intv $\otimes$ Strp. All the domain operations are lifted pair-wise to the product domain. Sometimes we may want to use the information contained in Intv to refine the information in Strp. For instance, to improve the precision of the join operator, as shown by the next example.

Example (Refinement of* Strp *with* Intv*) Consider the following piece of code:

```
int [] array;
...
// suppose that array.Length − count > 0
if (count == 0)
  array = new int[1];
else
  /* do nothing */ ;
```

Using *just* Strp, at the join point we cannot conclude that `array.Length` $-$ `count` > 0: inside the conditional, `array` is assigned a new value, so that the entry constraint is dropped.

Using Intv $\otimes$ Strp, the abstract state after array creation is $\bar{\mathtt{p}}_1 = \langle\langle \mathtt{count} \in [0,0], \mathtt{array.Length} \in [1,1]\rangle, \lambda \mathtt{z}.\ \emptyset\rangle$; the abstract state at the end of the false branch is $\bar{\mathtt{p}}_2 = \langle\emptyset, [\mathtt{array.Length} \mapsto (\mathtt{count},1,0)]\rangle$. The join is $\langle\emptyset, [\mathtt{array.Length} \mapsto (\mathtt{count},1,0)]\rangle$, as the interval component of $\bar{\mathtt{p}}_2$ implies that $\mathtt{array.Length} - \mathtt{count} > 0$. □

8.2 Compilation of `fixed`

When the C# compiler compiles a `fixed` statement which assigns an array `arr` of type `T[]` to a pointer `p`, it generates code to check whether the `arr` is `null` or if its length is 0. If it is the case, then it assigns `null` to `p`. Otherwise it assigns the address of the first element of `arr` to `p`. Fig. 12 depicts this compilation schema.

Without any refinement, the analysis performed by `Clousot` cannot capture that $\mathtt{WB}(\mathtt{p}) = \mathsf{sizeof}(\mathtt{T}) * \mathtt{array.length}$. Two main reasons for that: (1) it is not possible to represent a constraint in the form of $\mathtt{x} - a * \mathtt{y} = 0$ in Intv $\otimes$ Strp; (2) At the

```
  if (arr == null)
    p = null;
  else if (arr.Length == 0)
    p = null;
  else
    p = &arr[0];
```

Figure 12. The (schema of the) code generated by the C# compiler for the statement $\mathtt{fixed}(\mathtt{T} * \mathtt{p} = \mathtt{arr}) \ldots$ when `arr` is an array.

Assembly	# Methods	Time	# Accesses Checked	Validated	%
mscorlib.dll	18 084	3m43s	3 069	1 835	59.79
System.dll	13 776	3m18s	1 720	1 048	60.93
System.Data.dll	11 333	3m45s	138	59	42.75
System.Design.dll	11 419	2m42s	16	10	62.50
System.Drawing.dll	3 120	19s	48	29	60.42
System.Web.dll	22 076	3m19s	88	44	50.00
System.Windows.Forms.dll	23 180	4m31s	364	266	73.08
System.XML.dll	10 046	2m41s	772	311	40.28
Average					57.96

Table 3. The results of our analysis tested on the .NET assemblies without using any contract. The average analysis time is of 12ms per method.

join point, a state where p is null is merged with one where $\mathtt{WB(p)} = \mathsf{sizeof(T)} * \mathtt{array.length}$.

For (1), we refine the abstract domain to use LinEq, to retain linear equalities: the abstract domain used in the analysis becomes $\mathsf{LinEq} \otimes \mathsf{Intv} \otimes \mathsf{Strp}$.

For (2), if $\mathtt{arr} = \mathtt{null}$ or $\mathtt{arr.Length} = 0$, then $0 = \mathsf{sizeof(T)} * \mathtt{array.Length} = \mathtt{WB(p)}$ trivially holds. As we are performing an over-approximation of the reachable states, we can safely add $\mathtt{WB(p)} = \mathsf{sizeof(T)} * \mathtt{array.length}$ to our abstract state.

9. Experiments

We have implemented the analysis for unsafe memory accesses using the Stripes domain in Clousot. We have extensively tested our analysis on all the libraries of the .NET framework. Our experiments were conducted on a 2.4Ghz Intel Core Duo laptop, with 4Gbytes of RAM, running Windows Vista (Windows processor score 5.3). The target assemblies are taken from the %WINDIR%\ Microsoft\ Framework\ v2.0.50727 directory of the test laptop. No pre-processing, manipulation or filtering of the assemblies has been conducted.

A primary goal for Clousot is its use at development time during compilation or even within the integrated development environment. Thus, the performance of the analysis is crucial. Our specialized domains provide us with excellent performance as reported in Tab. 3.

The analysis is fast: the average analysis time per method is 12ms. We validate on average 57.96% of the unsafe memory accesses. This may not seem high at first glance. However, consider the burden of human code reviews for unsafe code which is currently a necessary practice. Our analysis cuts down the work load in half, focussing the reviews on accesses that seem non-obvious to prove correct. Nevertheless, we feel that we can improve the precision of the unsafe analysis in two ways:

1. We intend to remove short-comings in the current implementation of the domains, resulting in unnecessary precision loss or inability to prove facts that are implied. We intend to improve the domains as described e.g., in Section 7.3.
2. The code we analyzed does not contain contracts. This leads to loss of precision when the proof obligation required in one method is established by the caller of the method, or sometimes several call frames higher on the stack. As a consequence, without contracts on the intermediate methods Clousot reports warnings on those memory accesses.

We are actively working on adding contracts to eventually validate all memory accesses. Furthermore, to simplify checking of Windows API uses, we plan to write a tool to convert SAL annotations [15] into FoxTrot annotations. In Section 9.2 we discuss the results of manually inspecting the warnings for System.Drawing.dll and formulating necessary contracts.

9.1 Comparison with Polyhedra

The main claim of our work is that specialized domains targetting a particular set of proof-obligations are *required* to make such analyses practical. If we were able to use off-the-shelf solvers for more powerful domains, such as Polyhedra, specialized domains would not be necessary. We used our experience with the Polyhedra implementation used to infer loop invariants in Boogie [2], to evaluate the cost of using Polyhedra for the analysis of unsafe MSIL code. Although this implementation of Polyhedra is not as optimized as for example [1], it has been well debugged and in use for a number of years. In our experiment, we replaced the $\mathsf{Strp} \otimes \mathsf{Intv} \otimes \mathsf{LinEq}$ domain in our analysis with the Polyhedra domain implementation of Boogie and ran it on the two largest libraries in .NET. The results are shown in Table 4.

As is apparent from the timings, the Polyhedra domain is orders of magnitude slower than our implementation using Strp. In our runs, we used a 2 minute timeout per method. The timeout was reached 23 times on mscorlib.dll and 13 times on System.dll. In all fairness, the Parma library [1] is likely to be much faster than the implementation of Polyhedra we used. However, it is unlikely to consistently improve

Assembly	Time	# Accesses Checked	Validated	%
mscorlib.dll	125m52s*	3 070	1 610	52.46
System.dll	257m27s*	1 576	744	44.94

Table 4. Unsafe code analysis using the Polyhedra domain

the execution by two orders of magnitude and it would still suffer from exponential behavior on some methods where the 2 minute timeout was reached. When removing the timeout, one method in mscorlib.dll took 49 minutes to reach a fixpoint using Polyhedra.

9.2 System.Drawing case study

We analyzed the 19 warnings in System.Drawing.dll to determine what contracts need to be written to avoid them, or whether they represent true vulnerabilities.

First, we found the use of two helper methods that required pre-conditions:

```
short GetShort(byte* ptr) {
  Contract.Requires(Contract.WritableBytes(ptr)
                    >= sizeof(short));
  ...

int GetInt(byte* ptr) {
  Contract.Requires(Contract.WritableBytes(ptr)
                    >= sizeof(int));
  ...
```

These helper methods simply load 16 bits or 32 bits from the given pointer location using little-endian encoding and avoiding unaligned accesses.

With the pre-conditions written as above, Clousot no longer reports warnings within these helper methods. Instead, it reports warnings at 26 call-sites to these methods. The remaining warnings are all located within 5 distinct methods.

1. One method uses an unmanaged heap allocation routine to obtain memory from the marshal heap. Writing an appropriate post-condition for this allocator eliminates the warnings in that method.

   ```
   public static IntPtr AllocHGlobal(int cb) {
     Contract.Ensures(Contract.WritableBytes(
                      Contract.Result<IntPtr>()) == cb);
       ...
   ```

2. The next method we examined actually contained an error leading to buffer overruns on read accesses.
3. The third method uses a complicated invariant on a data structure that involves indexing using a product expression of two variables. Our domains cannot currently track such products (only variables multiplied with constants). However, the code appears to be safe.
4. The fourth method extracts a byte[] from an auxiliary data structure and indexes it assuming the array contains 1K elements. Examining the data structure and all its construction sites, we determined that it is built via marshalling from an unmanaged Windows API call and the marshal annotation specifies that the buffer is to be allocated with the fixed size of 1K. Although we can specify this size as an object invariant on the auxiliary structure leading to the removal of the warning by Clousot, our tool chain does not yet understand the marshalling constraints establishing the invariant.
5. Finally, the last function containing most of the accesses and calls to the helper functions GetShort and GetInt, whose pre-conditions must be validated, exposed a shortcoming in our implementation. Upon examination, we determined that the analyzer infers a sufficiently strong loop invariant which implies the safety of the memory accesses and pre-conditions. However, our implementation was not able to show this implication automatically.

With the above contracts and fixes, Clousot would validate 3 additional methods, but report false warnings in one method due to an index expression we cannot handle, and another false warning in a new method due to the lack of support for marshal annotations.

9.3 Summary

Overall, the analysis is fast enough to use in integrated development environments. It achieves a higher level of automation and scalability than existing tools. In fact, we found that the tool rarely fails to infer the necessary loop invariants to validate the memory accesses. More often, it is the lack of contracts that limits our modular intra-procedural analysis. The use of contracts not only allows reducing the false positive rate, the contracts furthermore serve as checked documentation on important safety invariants. Clousot can catch code changes or additions that fail to live up to the existing specifications and thereby provide excellent static regression checking.

10. Related work

In addition to the work cited in the introduction, we wish to place our work in the context of the following other related work.

Bounds analysis for C

Rinard and Rugina published a powerful analysis of C programs to determine aliasing, bounds, and sharing of memory, enabling bounds optimizations, and parallelization [28, 29]. Their analysis infers a set of polynomial bounds on variables that are solved using a linear programming problem to minimize the spread of the bound. The reported analysis times are fast (in the same range as ours), but they only report results for small examples. Their technique based on solving a linear programming problem is quite different from using

symbolic abstract domains, but equally promising. A benefit of their approach is that it performs inter-procedural analysis by inferring relations for function inputs and outputs using a bottom up call graph approach. However, this is also a major drawback, as for strongly connected components of functions (recursively calling each other), their analysis needs to compute a fixpoint. It is well known that call-graphs built for very large applications (in particular object-oriented programs) are imprecise, leading to very large components [11], making such an approach unlikely to scale.

Das et. al. describe buffer overflow checking and annotation inference on large Microsoft C/C++ code bases [15]. Few details of the used numerical domains are public, but from the paper it is apparent that for precision, their analysis performs path splitting, meaning it analyzes paths separately through a function whenever the abstract state at join points disagrees. The Stripes domain described in this paper and the associated transfer functions and join operations are geared towards providing precision without path splitting (our analyzer does not perform path splitting).

Analysis of JNI

A few analyses for Java handle programs using the Java Native Interface (JNI) [20]. Furr and Foster in [14] present a restricted form of dependent types used to infer and type-check the types passed to foreign functions via the JNI. Tan *et al.* proposed a mixed dynamic/static approach to guarantee type safety in Java programs that interface with C. We are not interested in type safety: in unsafe C#, type errors are less common than with the JNI, since the unsafe context is integrated in C#, so that (a) the compiler can still perform most type checking and (b) types do not need to be serialized as strings (the most common type error in using the JNI). Instead our analysis focuses directly on memory usage via pointers, whereas previous work did not.

Interoperability of languages

Recent work focuses on language interoperability. Tan and Morrisett, [33], advocate an approach in which the Java bytecode language is extended with a few instructions useful to model C code. Hirzel and Grimm, [16], take an alternative approach with Jeannie, which is a language which subsumes Java and C, and the burden of creating the "right" JNI for interfacing the two languages is left to the compiler. Matthews and Findler, [24], give an operational semantics for multi-language programs which uses contracts as glue for the interoperating languages. The `MSIL` instruction set is rich enough to allow an agile compilation of several languages: our analysis, working at the `MSIL` level does not need to take into account inter-operability issues.

Static analyzers

ESC/Java 2 [6] and Spec# [4] use automatic theorem provers to check programs. Automatic theorem provers provide a strong engine for symbolic reasoning (*e.g.* quantifiers handling). The drawbacks are that: (a) they require the programmer to provide loop invariants and (b) they present scalability problems. Analysis times close to the one we obtain in `Clousot` on shipped code are well beyond the state-of-the art in automatic theorem proving.

11. Conclusions

We presented a new static analysis for checking memory accesses in unsafe code in .NET. The core of the analysis is a new abstract domain, Strp, which combined with Intv and LinEq, allows the analysis to scale to hundreds of thousands of lines of code. We have proven the soundness of the approach by designing the static analysis using stepwise abstraction of a concrete transition semantics.

References

[1] R. Bagnara, P.M. Hill, and E. Zaffanella. The Parma Polyhedra Library. `http://www.cs.unipr.it/ppl/`.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for Object-Oriented programs. In *FMCO'05*. Springer-Verlag, November 2005.

[3] M. Barnett, M. Fähndrich, and F. Logozzo. Foxtrot and Clousot: Language Agnostic Dynamic and Static Contract Checking for `.Net`. Technical Report MSR-TR-2008-105, Microsoft Research, Redmond, WA, August 2008.

[4] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, 2004.

[5] G. P. Brat and A. Venet. Precise and scalable static program analysis at NASA. In *IEEE Aerospace Conference*. IEEE, 2005.

[6] D. R. Cok and J. Kiniry. ESC/Java 2: Uniting ESC/Java and JML. In *CASSIS 2004*, 2004.

[7] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM Press, January 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*, pages 269–282. ACM Press, January 1979.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*. ACM Press, January 1978.

[11] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, pages 35–46. ACM, 2000.

[12] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *SAS'01*, LNCS. Springer-Verlag, June 2001.

[13] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI'03*. ACM Press, 2003.

[14] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *ESOP'06*. Springer-Verlag, April 2006.

[15] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ACM ICSE'06*. ACM Press, 2006.

[16] M. Hirzel and R. Grimm. Jeannie: granting Java native interface developers their wishes. In *OOPSLA'07*. ACM, October 2007.

[17] R. N. Horspool and J. Vitek. Static analysis of postscript code. In *ICCL'92*. IEEE, 1992.

[18] M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, July 1976.

[19] L. Khachiyan, E. Boros, K. Borys, K. M. Elbassioni, and M. Gurvich. Generating all vertices of a polyhedron is hard. In *ACM SODA'06*. ACM Press, 2006.

[20] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Sun Microsystems, 2001.

[21] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *VMCAI'07*. Springer-Verlag, January 2007.

[22] F. Logozzo and M. A. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC'08*, LNCS. Springer-Verlag, March 2008.

[23] F. Logozzo and M. A. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *ACM SAC'08 - OOPS*. ACM Press, March 2008.

[24] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL'07*. ACM, January 2007.

[25] B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.

[26] A. Miné. The octagon abstract domain. In *WCRE 2001*. IEEE Computer Society, October 2001.

[27] M. Müller-Olm and H. Seidl. A note on karr's algorithm. In Springer-Verlag, editor, *ICALP'04*, LNCS, 2004.

[28] R. Rugina and C. R. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, volume 35.5 of *ACM Sigplan Notices*, pages 182–195, N.Y., June 18–21 2000. ACM Press.

[29] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, 27(2):185–235, 2005.

[30] D. A. Schmidt. The internal and external logic of abstract interpretations. In *VMCAI'08*. Springer-Verlag, January 2008.

[31] A. Simon and A. King. Analyzing string buffers in c. In *AMAST'02*, LNCS. Springer-Verlag, September 2002.

[32] A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*, LNCS. Springer-Verlag, September 2002.

[33] G. Tan and G. Morrisett. Ilea: inter-language analysis across java and c. In *OOPSLA'07*. ACM, October 2007.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS'00*, 2000.