Mind your Language(s)

A discussion about languages and security (Long Version)

Éric Jaeger, Olivier Levillain and Pierre Chifflier
Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)
firstname.lastname@ssi.gouv.fr

Nota: This report is an extended version of a paper published in the *LangSec* workshop proceedings, a satellite event of the *IEEE Symposium on Security and Privacy* 2014. Please refer to http://spw14.langsec.org/ for further information.

***Abstract*—Following several studies conducted by the French Network and Information Security Agency (ANSSI), this paper discusses the question of the intrinsic security characteristics of programming languages. Through illustrations and discussions, it advocates for a different vision of well-known mechanisms and is intended to provide some food for thoughts regarding languages and development tools, as well as recommendations regarding the education of developers or evaluators for secure software.**

> An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.
>
> C.A.R Hoare

The French Network and Information Security Agency (ANSSI[1]) is an organism whose mission is to raise the security level of IT infrastructures to the benefit of governmental entities, companies as well as the general public. This includes identifying, developing and promoting methods or tools e.g. to improve correctness and robustness of software.

In this area, the ANSSI has been conducting for a few years different studies on the adequacy of languages (including formal methods) for the development of secure or security applications. The objective of these studies was not to identify or specify the best possible language, but to understand the pros and cons of various paradigms, methods, constructions and mechanisms. Working for some of these studies with language specialists and industrial users, one of the first lessons learned was that there was no common understanding about this notion of intrinsic security of languages, and furthermore that some of the concerns raised by the security experts were not understood at first by the other actors.

This report is intended to shed light on those concerns, as exhaustively as possible, through numerous illustrations. It attempts to summarise our journey among programming languages, identifying interesting features with potential impacts on the security of applications, ranging from language theory to syntactic sugar, low-level details at runtime, or development tools. Far from being conclusive, we expect no more than to promote, as far as security is a concern, alternative visions when dealing with programming languages; our professional experience is that this discussion is definitively useful today[2].

The subject of this article is therefore quite broad, and to some extent difficult to structure in an appropriate manner for the different communities. We have chosen to go from language theoretic aspects to low-level concrete details, then to discuss additional questions related e.g. to evaluation. After a short discussion about security, Sec. II considers abstract features and paradigms of languages. We will then take a look under the hood, discussing syntax and more generally the concrete constructions provided by languages, that can have a noticeable impact on the quality of (or the confidence in) programs; this is the subject of Sec. III. Next, Sec. IV cast some light on those steps leading from source code to a running program that definitely deserve proper attention, such as the compilation phase and the execution environment. Once developed, a product has still to be tested and evaluated to derive an appropriate level of assurance, as it is discussed in Sec. V. Finally, Sec. VI ends this paper with a few lessons learned and perspectives.

I. Language security

A secure software is expected to be able to cope with wilful aggressions by intelligent agents. In this sense, security approaches can significantly differ from functional ones (when the objective is e.g. to ensure that for standard inputs the software returns the expected results) or safety and dependability ones. In any case, bugs have to be tracked and eradicated, but securing a software may require additional care, trying to prevent unacceptable behaviors whatever the circumstances (and regardless of their probability of occurrence).

Consider for example a program for compression and decompression. The functional specification of Compress and Uncompress is that for any file f we have:

$$\mathsf{Uncompress}(\mathsf{Compress}(f)) = f$$

However, it says nothing about the behavior of Uncompress when applied to arbitrary inputs. Those can be the result of errors, but also maliciously crafted files, which are a common security concern (e.g. CVE-2010-0001 for `gzip`). In essence,

[1] http://www.ssi.gouv.fr

[2] That is in particular following recent publications about software bugs leading to severe security vulnerabilities in major SSL/TLS stacks, such as APPLE's one (*Goto Fail*), GNUTLS (*Goto Fail*) and OPENSSL (*Heartbleed*).

we would also like to see the dual specification for resilience, that is for any file c we have:

$$\neg(\exists\, f,\, \mathsf{Compress}(f) = c) \Rightarrow \mathsf{Uncompress}(c) = \mathbf{Error}$$

We are also interested in security software, that is products offering security mechanisms. Such software has to be secure but has also to meet additional requirements to protect data or services. A good example discussed thereafter is a library providing cryptographic services, which is expected to protect the cryptographic keys it uses, preventing unauthorised modifications or information leaks[3].

Finally, we have to take care of the question of the evaluation. Indeed, a secure software may not be of any use in practice if there is no way to know that it is secure. It is therefore important to have processes to obtain appropriate assurance levels (see for example the Common Criteria standard [CC]).

There exist numerous works concerning language safety, and to some extent one may consider that languages such as ERLANG[4] or SCADE[5] have been designed specifically to cope with resilience, safety or dependability. Many general purpose formal methods can also be seen as tools to adequately address functional or safety needs. However, to our knowledge, the question of the security of languages is not so popular, and literature is still rather scarce. To be fair, there are books providing recommendations to develop secure applications in a given language, as well as proposals to natively include security features such as e.g. information flow control in languages [HA05]. But from our point of view, the security concerns are much broader than that, and should be addressed not only through coding recommendations or other software engineering methods, but also by due consideration during design phases of languages and associated tools.

This has led ANSSI to conduct several works dealing with security and development, looking at several programming languages or development methods with a critical (and admittedly unfair) eye. This includes a first study to better understand the contributions of the JAVA language [ANS10], a second study on functional languages in general and OCAML in particular [ANS13], as well as a review of the benefits and limits of formal methods [JH09], [Jae10].

In the remaining of this paper, we summarise many of our concerns through illustrations, to give some food for thoughts for the academic communities interested by languages. We also expect to promote a different vision of what a programming language for developers interested by security concerns is, as well as recommendations on education and training.

The sources of our concerns are numerous – and not always subtle: they range from syntax traps or false friends for inattentive developers to obfuscation mechanisms allowing a malicious developer for hiding a backdoor in a product while escaping detection by an evaluator, but also include theoretical properties not enforced in executable code or inappropriate specifications.

[3]Techniques used to protect cryptographic implementation against side-channel attacks [CJRR99], [GP99] are out of the scope of this paper.

[4]http://www.erlang.org

[5]http://www.esterel-technologies.com/products/scade-suite

Code excerpts presented in the following sections use different programming languages. Our intent is not to criticise a specific language, but to illustrate concrete instances of our concerns (more often than not, the concepts are applicable to other languages). Furthermore, we may provide negative reviews for some mechanisms or concepts; the reader is expected to remember that we only deal in this paper with security concerns for critical developments, and that our messages should not be generalised too quickly to other domains.

Acknowledgement: We would like to mention that some of our illustrations have been derived from examples on different websites and blogs such as [Koi], [Atw], [FO] and (last but not least) [DW].

II. ABSTRACT FEATURES AND PARADIGMS

> The tools we are trying to use and the language or notations we are using to express or record our thoughts, are the major factors determining what we can think or express at all!
>
> Edsger W. Dijkstra

A. Scoping and Encapsulation

We start our discussion with a family of features existing in nearly any mainstream languages, namely the scoping of identifiers and the encapsulation. They are rather simple mechanisms, and such information is helpful e.g. for the compiler to define an appropriate mapping or to apply possible optimisations. Furthermore, they are attractive for the developer dealing with security objectives, e.g. to protect confidentiality or integrity of data. We believe that developers rely on many assumptions about frontiers between various parts of software – they expect proper separation between e.g. local variables of an application and a library it uses, and *vice versa*. Any mechanism blurring such frontiers is likely to translate into security concerns.

Variable scoping: In general, languages come with variables of different sorts, in terms of scope (local *vs* global) and life cycle (constant, variable or volatile). This is closely related to various theoretical concepts such as bound variables and α-renaming, and is so common that it has become part of developers' intuition.

But for a few languages, the design choices appear rather unexpected, especially when syntax does not provide any clue. Consider the following snippet in PHP:

```
$var = "Hello World ";
$tab = array("Foo ", "Bar ", "Blah ");
{ foreach ($tab as $var) { printf($var); } }
printf($var);
```

Code snippet 1.

The `foreach` loop enumerates the values of the array `$tab` and assigns them to the variable `$var`. This code prints successively `Foo`, `Bar`, `Blah`... and `Blah` at its last line: the variable `$var` in the `foreach` loop is not locally bound and its previous value `Hello World` is overwritten.

PYTHON has a similar behavior with the comprehension list construct, e.g. in `[s+1 for s in [1,2,3]]` (that yields list `[2,3,4]`) the variable `s` should be locally bound, but survives

the definition. This is unexpected, and in fact inconsistent with other similar constructs in PYTHON such as the `map` construct[6].

This is, in our view, sufficiently unexpected and intriguing to be dangerous. Developers rely on compositionality, and such poor managements of scopes mean that the semantics of closed pieces of code – using only locally defined variables – now depend upon their context.

Encapsulation: The encapsulation, e.g. for object-oriented languages, is a form of extension of scoping, but with a different flavor. In JAVA for example, one can mark a field as `private` to prevent direct access except from other instances of the same class. Problems arise when a developer confuses this software engineering feature with a security mechanism.

Consider this example in JAVA, in which a class `Secret` has a private field `x`:

```
import java.lang.reflect.*;

class Secret { private int x = 42; }

public class Introspect {
  public static void main (String[] args) {
    try { Secret o = new Secret();
          Class c = o.getClass();
          Field f = c.getDeclaredField("x");
          f.setAccessible(true);
          System.out.println("x="+f.getInt(o));
        }
    catch (Exception e) { System.out.println(e); }
  }
}
```

Code snippet 2.

The printed result is `x=42`: introspection is used to dynamically modify the class definition and remove the `private` tag.

It is possible to forbid the use of introspection in JAVA with the so-called security monitor, yet this is a rather complex task – which is furthermore likely to have side effects on standard library services, as for example serialization in JAVA uses introspection.

OCAML provides encapsulation mechanisms through objects, but also a more robust version based on modules, whose public interface can be partial with respect to their actual implementation. For example, the following `C` module exports its `id` field but not `key`, as specified by its interface `Crypto`:

```
module type Crypto = sig val id:int end

module C : Crypto  =
struct
  let id = Random.self_init(); Random.int 8192
  let key = Random.self_init(); Random.int 8192
end
```

Code snippet 3.

That is, `C.id` is a valid expression whereas `C.key` is not, being rejected at compilation time.

This is a pretty interesting feature to protect data, relying on well-studied type checking mechanisms: from the academic perspective, such a module is a closed box that cannot be opened. Yet there are in OCAML literature descriptions that do not fit together. In [WL99], for example, a polymorphic function is described as a function that does not analyse the whole structure of its parameters, whereas the reference manual indicates that functions such as `<` are polymorphic and compare the structure of their parameters. Enters version 3.12 of OCAML, introducing first-class modules (*i.e.* modules are values that are comparable).

This led us to write this little experiment:

```
let rec oracle o1 o2 =
   let o = (o1 + o2)/2
in let module O = struct let id=C.id let key=o end
in if (module O:Crypto)>(module C:Crypto)
   then oracle o1 o
   else if (module O:Crypto)<(module C:Crypto)
   then oracle o o2
   else o
```

Code snippet 4.

The function `oracle` is parameterised by `o1` and `o2`, the lower and upper bounds. It creates a module `O` with `key` set to `o`, the mean value of the bounds, and compare it with `C`. If `O>C`, the function is invoked again replacing the upper bound by `o` (or the lower bound if `O<C`). In practice, the `oracle` function finds the hidden value of `C.key`[7]. The attack is efficient (logarithmic in time) but the main point is that we have been able to cross module frontiers using standard language constructs.

Adopting again the academic perspective, the box has not been opened, but its contents is revealed by using a weighing scale. Some consider that this argue against polymorphic comparison operators in OCAML; for us it shows that trusted theoretical notions are not automatically and easily translated into robust security properties.

Our comments regarding encapsulation mechanisms are not intended to pinpoint errors in the languages: such mechanisms are convenient design tools of software engineering. On the other hand, it should be clear for developers that they are not, in general, security mechanisms.

B. Side effects

A fundamental result of typed λ-calculus (a pure functional language) states that evaluation strategy has no influence on the result of computations. Conversely, in presence of side effects, the order of computations may become observable.

Provided this notion of evaluation strategy does not appear to be part of developer's common knowledge, we may expect some confusion when dealing with side effects. In [KR88] for example this is explicitly addressed by a clear and simple explanation of the difference of behavior between the following macro and function when one computes `abs(x++)`:

```
#define abs(X) (X)>=0?(X):(-X)
```

Code snippet 5.

```
int abs(int x) { return x>=0?x:-x; }
```

Code snippet 6.

Yet we would like to discuss more intriguing situations:

```
{ int c=0; printf("%d %d\n",c++,c++); }
{ int c=0; printf("%d %d\n",++c,++c); }
{ int c=0; printf("%d %d\n",(c=1),(c=2)); }
```

Code snippet 7.

Well-informed C developers guess that the first line prints `1 0` as a consequence of right-to-left evaluation of parameters for the call-by-value strategy, but are generally surprised that the second line prints `2 2`.

[6]The behavior of comprehension lists has been fixed in PYTHON 3 but not in PYTHON 2, for the sake of backward (bugward?) compatibility.

[7]The complete explanation of why it works also relies on the existence of a not-so-appropriate compiler optimisation.

At this stage, it would be cautious to admit that the pre- and post-increment operators can be too subtle to use. And, as in practice we will not miss them badly, why do they exist at all in the language?

We are not over yet with side-effects, as affectations are by definition the primitive form of side-effect, and as a bonus are also a value in many languages such as C – a cause of troubles for generations of developers that have accidentally inserted an assignment instead of a boolean test in their `if` statements. The third line of the example therebefore prints `1 1` (this is also the final value of the variable `c`), something not so easily explained except by discoursing on calling conventions of C. This type of code is confusing and explicitly discouraged by C standards: why then is it compiled without even a warning?

By extension, *anything* revealing the evaluation strategy can be considered as a side effect. This goes beyond assignments, and also includes for example errors or infinite loops.

Let us play again with macros and functions. What would be the difference between the two following versions of `first`?

```
#define fst(X,Y) X
```
Code snippet 8.

```
int fst(int x,int y) { return x; }
```
Code snippet 9.

As previously, they can be distinguished e.g. with expressions such as `first(x,y++)`. But there are other methods and one can reveal which one is used with `fst(0,1/0)`, as the call-by-value strategy for functions will throw an exception.

Pretty straightforward, so what do the following pieces of code (without even playing with macros)?

```
int zero(int x) { return 0; }
int main(void) { int x=0; x=zero(1/x); return 0; }
```
Code snippet 10.

```
int zero(int x) { return 0; }
int main(void) { int x=0; return zero(1/x); }
```
Code snippet 11.

There is unfortunately no simple answer to this simple question: using the GCC compiler, there is an exception for both with option `-O0`, with option `-O1` the first code returns `0` and with option `-O2` the second also returns `0`.

The fact that standard optimisations[8] can modify the observable behavior of a program is worrying, and reveals in our view a lack of clear and non ambiguous semantics for the C language.

As a last comment about side effects, let us go back to more basic observations but with unexpected applications to OCAML. In this language any standard variable is in fact a constant, in the sense that it can only be declared, allocated and initialised at once, and that it cannot be assigned later. Yet all allocations are managed by the garbage collector and variables live on the writable heap – this prevents the use of low-level security mechanisms such as storing constants in read-only pages for example.

OCAML also provides mutable strings (it is an impure functional language). Put together, this actually means that there is no way to have constant strings, which allows for dirty tricks such as this one:

```
let alert = function true -> "T" | false -> "F";;
(alert false).[0]<-'T';;
alert false;;
```
Code snippet 12.

The first line of this code declares a function `alert` with a boolean parameter and returns a string, either `"T"` for `true` or `"F"` for `false`. The second line computes `alert false` to get the reference to the `"F"` string and overwrites its first char with `'T'`. As a consequence, the third line returns the string `"T"` instead of `"F"` – as any further invocation of `alert false`.

Again, this is logical, yet surprising: as far as the developer is concerned, the source code of the function `alert` appears to have been modified by a side effect.

This also applies to standard library functions of OCAML (at least in version 3.12 of the language[9]) and one can modify for example `string_of_bool` – by the way also impacting the behavior of `Printf.printf`. Similarly, it is a common practice in OCAML to parameter exceptions with strings, that can later be pattern-matched to make decisions; modifying such strings can then interfere with execution flow.

As a final example, let us mention that the strings returned by `Char.escaped`, which is a security mechanism, can also be meddled with. The following code is a program supposed to call `external_interpreter` with each of its arguments. Following best practices, strings are first escaped using `Char.escaped` on each character:

```
let escape_string req =
  let n = String.length req in
  let clean_req = Buffer.create (2 * n) in
  for i = 0 to n - 1 do
    Buffer.add_string clean_req (Char.escaped req.[i])
  done;
  Buffer.contents clean_req

let main () =
  for i = 1 to ((Array.length Sys.argv) - 1) do
    external_interpreter (escape_string Sys.argv.(i))
  done
```
Code snippet 13.

However, as escaped strings returned by `Char.escaped` are mutable, executing e.g. `(Char.escaped '"').[0]<-'.';` change the observable behavior of the program, which will incorrectly escape "'" as "." instead of "\'", which may lead to security vulnerabilities. As with previous examples, it is disturbing to have a statement located anywhere in a program changing the behaviour of a standard library function.

C. Types

> Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime literally impossible, because there will be no words in which to express it.
>
> 1984, George Orwell

In Mathematics, type theory was a proposal of *Bertrand Russel* to amend *Gottlob Frege*'s naive set theory in order to avoid *Russel*'s paradox. In computer science, type checking is a well-discussed subject, and provides a good level of assurance with respect to the absence of whole categories of bugs – especially when the associated theory is proven. It

[8] Admittedly, the `-O3` flag is known to have curious effects and its use is not recommended, but as far as we knew this was not the case with `-O2`.

[9] Actions have been taken since then to avoid such manipulations e.g. by systematically returning copies of strings instead of the original ones.

can statically reject syntactically valid but meaningless expressions, but can also enforce encapsulation, manage genericity or polymorphism, and so on.

In the family of ML languages [Mil84], [MT91], such as OCAML, it also leads to type inference and static verification of the completeness and relevance of pattern matching constructs, actually providing great assistance to developers. Type-checking is therefore relatively powerful, efficient and, last but not least, it does not contradict developers' intuition (yet advanced type systems may induce subtle questions, e.g. when dealing with subtypes and higher-order constructs [Pie02]). Thus we consider that type-checking is a must for secure developments, preferably both static and strong to ensure early detection of ill-formed constructs.

Casts and overloading: It is often considered that strict application of type-checking concepts leads to cumbersome coding standards, not-so-friendly to developers. OCAML for example distinguishes between integer addition `+` and floating-point addition `+.` and does not automatically coerce values. The expression `1. + 2` is therefore rejected, one correct version being `1. +. (float_of_int 2)`.

It is therefore standard for languages and compilers to ease developer's work by providing automated mechanisms allowing for overloading (using the same identifier for different operations) and automated casts or coercions. Yet we have various concerns about this approach. A first and trivial comment is that, whereas every student uses this type of trick, nearly none of them can explain what's going on. Casts and overloading are not a comfort anymore but a disguise.

> All animals are equal, but some animals are more equal than others.
>
> Animal Farm, George Orwell

But we are also worried by the consistency of design choices. Consider the example of ERLANG, in which expressions such as `1+1`, `1.0+1.0` and `1+1.0` are valid, implicitly inviting developers not to care about the difference between integers and floats. Let us now consider the typical example of the factorial function, presented in all beginner lessons:

```
-module(factorial).
-compile(export_all).
fact(0) -> 1;
fact(N) -> N*fact(N-1).
```

Code snippet 14.

Without surprise, `factorial:fact(4)` returns `24`, but on the other hand `factorial:fact(4.0)` causes a stack overflow.

The same type of remarks apply to JAVASCRIPT, in which the condition `0=='0'` (comparing an integer with a string) is true, but a `switch (0)` does not match with `case '0'`.

Beyond such inconsistencies, one can also wonder whether casts and overloads are worth missing fundamental and intuitive properties.

For example, equality is expected to be transitive, but this is not the case in JAVASCRIPT as `'0'==0` and `0=='0.0'` are true but `'0'=='0.0'` is false. Similarly `+` can represent integer addition, floating point addition or string concatenation. Consider now the following example:

```
a=1; b=2; c='Foo';
print(a+b+c); print(c+a+b); print(c+(a+b));
```

Code snippet 15.

This code prints `3Foo`, `Foo12` and `Foo3`. Whereas all the operations represented by `+` are associative, the property disappears for the composite (overloaded) operator.

Let us continue our discussion on casts and overloading with examples in the C language. Here again, It is notorious that unexperimented developers can be tricked, the canonical example being:

```
int x=3;
int y=4;
float z=x/y;
```

Code snippet 16.

Of course, it results in `z` being assigned float value `0.0`.

This one is maybe a little too easy to explain, so consider the following one, without plays on types:

```
unsigned char x=128;
unsigned char y=2;
unsigned char z=(x*y)/y;
```

Code snippet 17.

Some may be surprised to find out that `z` is assigned `128` and furthermore disappointed that compiler's optimisation has nothing to do with this result: even in this code where all values have the same type, there are implicit casts.

The cast mechanism in itself can be quite subtle too, as the following code prints `1>=-1` and `1<-1`:

```
{ unsigned char a = 1; signed char b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }

{ unsigned int a = 1; signed int b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }
```

Code snippet 18.

At this stage, we have to consider coercions and overloadings as false friends, on some occasions too devious to be managed properly. Are we too severe? Well, in practice, this leads to situations such as the following one, in which a `safewrite` function has been specifically designed to check that writes in an array are valid with respect to its bounds:

```
#include <stdio.h>

void safewrite (int tab[],int size,
                signed char ind,int val) {
  if (ind<size) tab[ind]=val;
  else printf("Out of bounds\n");
}

int main(void) {
  size_t size=120;
  int tab[size];
  safewrite(tab,size,127,0);
  safewrite(tab,size,128,1);
  return 0;
}
```

Code snippet 19.

If `safewrite(tab,size,127,0)` indeed correctly produces the expected `Out of bounds` message, on the other hand `safewrite (tab,size,128,0)` succeeds – *why* and *where* the write occurs is left to reader's wisdom, as how would the program behave with `size=150` instead of `size=120`.

Of course some compiler options (such as `-Wconversion` for GCC) help to pinpoint such problems, but in practice there is a long history of bugs which are, in essence, similar to this one, leading to critical security vulnerabilities. This was the case for example with CVE-2010-0740, a really subtle bug in OPENSSL fixed by an even more subtle short patch:

```
-  /* Send back error using their
-   * version number :-) */
-  s->version=version;
+  if ((s->version & 0xFF00) == (version & 0xFF00))
+  /* Send back error using their minor version number */
+  s->version = (unsigned short)version;
```
Code snippet 20.

In the light of the previous examples, it might be hard to explain the exact consequences of inserting an explicit integer cast, whose effects can vary across architectures and compilers.

The JAVA language also allows developers for shooting themselves in the foot with overloadings as in this example, which prints `Foo`, `Bar` and `Foo`:

```
class Confuser {
  static void A(short i) { System.out.println("Foo"); }
  static void A(int i) { System.out.println("Bar"); }

  public static void main (String[] args) {
    short i=0; A(i); A(i+i); A(i+=i);
  }
}
```
Code snippet 21.

PHP induces a whole new category of difficulties as far as types are concerned. One can play for example with string arithmetics, and increment with `++` a variable storing the string `"2d8"`. The variable value is successively the `"2d9"` string, the `"2e0"` string and finally the `3` float, as `"2e0"` is interpreted as a scientific notation for $2 \cdot 10^0$!

The confusion between strings and numerical values is consistently reflected by the comparison operator `==`, which can induce casts even when comparing values of the same type. For example, the `"0xf9"=="249e0"` condition yields true as it does not compare the two strings but convert them respectively into `int(249)` and `float(249)`.

Let us illustrate possible consequences of these mechanisms with the following piece of code:

```
$h1=md5("QNKCDZO");
$h2=md5("240610708");
$h3=md5("A169818202");
$h4=md5("aaaaaaaaaaumdozb");
$h5=sha1("badthingsrealmlavznik");

if ($h1==$h2) print("Collision\n");
if ($h2==$h3) print("Collision\n");
if ($h3==$h4) print("Collision\n");
if ($h4==$h5) print("Collision\n");
```
Code snippet 22.

When executed, it prints `Collision` four times. It is difficult to believe that we have indeed four distinct short strings with the same MD5 hash, and impossible to have a collision between an MD5 hash and a SHA1 hash. The trick is that each of the computed hash is itself a string whose pattern matches the $[0]^+e[0-9]^+$ regular expression. When compared with `==` they are therefore all converted into the same value, that is `float(0)`.

To conclude this discussion, remember that we have presented type checking as a way to detect ill-formed expressions. To some extent, casts and overloadings weaken this detection, the compiler being authorised to manipulate the code until it has a meaning. But one should avoid a situation in which any syntactically valid construct would become acceptable!

JAVASCRIPT seems to be well advanced on this road, as all the lines in the following example are valid and have a distinct meaning, including the first and fourth one[10]:

```
{} + {}      // NaN
[] + {}      // "[object Object]"
{} + []      // 0
({} + {})    // "[object Object][object Object]"
```
Code snippet 23.

Type abstraction: Types provide a level of abstraction of programs convenient for the developers, but can lead to oversimplistic analyses in some cases.

As a first example, consider the following SQL code:

```
SELECT CONCAT(IF(@X<=@Y,'X<=Y','X>Y'),
              ' and ',
              IF(@X>=@Y,'X>=Y','X<Y')) AS Test;
```
Code snippet 24.

With `SET @X=1; SET @Y=2;` this code prints `X<=Y and X<Y`, as expected. But SQL also allows for representing the absence of value. With `SET @X=NULL;` we got `X>Y and X<Y` – a situation one may not expect when reading the code. The same type of comments can be made about the `NaN` value for floats in many languages.

Similarly, one can consider that a well-typed boolean expression will return either true or false. We would however argue that other behaviors are possible and should be considered as well, for example looping computations, crashes... or errors. Indeed, short-sighted view can easily become a cause of concerns, as illustrated here:

```
#!/bin/bash
PIN_CODE=1234
echo -n "4-digits PIN code for authentication: "
read -s INPUT_CODE; echo

if [ "$PIN_CODE" -ne "$INPUT_CODE" ]; then
    echo "Invalid PIN code"; exit 1
else
    echo "Authentication OK"; exit 0
fi
```
Code snippet 25.

This script checks whether a PIN code is valid or not for authentication. One can conclude that if the provided code is `1234`, authentication will succeed, whereas any other value is rejected. However, in practice the test relies on a shell command which is expecting numerical values. Should a non-numerical value such as `blah` be provided, this test would return an error code interpreted as being not true by the `if` statement, resulting in undue authentication.

In March 2014, an interesting vulnerability has been discovered in the GNUTLS library; let us just quote the analysis provided on LINUX Weekly News[11]:

> [This bug] has allowed crafted certificates to evade validation check for all versions of GNUTLS ever released since that project got started in late 2000.[...] The `check_if_ca` function is supposed to return true (any non-zero value in C) or false (zero) depending on whether the issuer of the certificate is a certificate authority (CA). A true return *should* mean that the certificate passed muster and can be used further, but the bug meant that error returns were misinterpreted as certificate validations.

As a matter of fact, a very similar issue, CVE-2008-5077, had been diagnosed in OpenSSL in 2008. The corresponding patch simply replaces a `if (!i)` line by `if (i <= 0)`.

[10] See http://jscert.org.

[11] http://lwn.net/Articles/589205/

Types at runtime: As a final remark about types, one should note that in static type-checking systems, types are pure logical information that have no concrete existence in implementations. This topic and its practical consequences are discussed in Sec. IV.

D. Evaluators

Some languages offer mechanisms to dynamically modify or create code, e.g. relying on evaluators. This allows for meta-programming and other dynamic features.

This is the case of the `eval` command in PHP, transforming (dynamically produced) strings into code which is executed. Other dynamic features exists in PHP. For example `$$x` refers to the variable whose name is stored in `$x` and `$x()` invokes the function whose name is stored in `$x`.

As a related subject, we will discuss in Sec. IV the notion of attack by injection. For now, let us note that as far as security is concerned, the use of any form of evaluator makes a program impossible to analyse: in our view, language-embedded evaluators forbid security evaluation.

III. Syntax and syntactic sugar

The elements presented in the previous sections deal with relatively high-level concepts. However, on occasions, the syntax of the language can be confusing or even misleading either for developers or evaluators. We provide a short review of concrete details that can become important.

A. Consistency

> Come, let us go down and confuse their language so they will not understand each other.
>
> Genesis 11:7

Identical keywords in different languages can have different semantics – something that we need to live with, but deserve adequate consideration e.g. for education of developers or evaluators.

But it may also be that the same keyword or concept is used with several semantics in a language, depending upon the context. One can consider for example in JAVA the various possible meanings of the `static` tag, or the use of the interface concept as a flag to enable some mechanisms provided by the standard library (e.g. `serializable`). We find such design choices confusing and therefore potentially dangerous.

B. No comments

As noted in Sec. II, the `x=1` assignment is also an `int` value in C, whereas `x==1` is a boolean condition (that is an `int` value). Confusing the two constructs is a common mistake, due to syntax similarity and the absence of warning from the compiler using standard options.

This can also be a trick used by a malicious developer, as in this now classical example[12] of what appears to be an attempt to insert a Trojan horse in the LINUX kernel:

```
+ if ((options==(__WCLONE|__WALL)) && (current->uid=0))
+ retval = -EINVAL;
```

Code snippet 26.

This small insert in the code of a system call mimics an error check. Yet in practice, should the two options `__WCLONE` and `__WALL` be set together (an irrelevant combination), and only in this case due to the lazy evaluation of `&&`, then the process user id would become `0`, that is `root`. This trap is discrete both with respect to syntax and behavior.

> Don't get suckered in by comments, they can be terribly misleading.
>
> Dave Storer

Some other apparently irrelevant details of syntax may also be misleading to reviewers, e.g. the very basic concept of comments. Modern C compilers for example allows for different types of comments, `//` to comment the end of a line and (C++ style) `/* */` to comment a block of code. Problems arise when a program contains the sequence `//*`, as different tools can have different interpretation of this program.

To consider another language, in OCAML comments are surrounded by `(*` and `*)`, can be nested, but also have an intriguing feature. Consider the following piece of code:

```
(* x" enable security checks *)
let x''=true;;

(* REMOVED ON Friday 13th 2013 ===============
(* x"=false to disable checks during tests *)
let x''=false;;
(* Set x"=true once tests are completed *)
   CHANGED x" TO ENABLE SECURITY CHECKS =========*)
```

Code snippet 27.

At first, it seems that the line `let x''=true` is executed, the rest of the code being commented out. But it is possible, in OCAML, to open a string in a comment – and this is exactly what we have done in our example. Therefore `let x''=true` is in fact commented out, whereas `let x''=false` is not. This is especially misleading when syntax coloring tools do not apply the appropriate rules (as this is indeed the case for some).

Similar tricks are possible in C, such as in this example:

```
// /!\ Do not uncomment /!\
/*********************************
const char status[]="Unsafe";
// /!\ Only for tests /!\
 *********************************/

// /!\ Important, do not remove /!\
const char status[]="Safe";
```

Code snippet 28.

Of course, it assigns `Unsafe` to the string `status`[13]. This can be used by a malicious developer to trick an evaluator, but it may also mean that, either because of a genuine ambiguity or because of misunderstandings, again different tools can have different interpretations for the same source file, a perspective we are worried about.

C. Encoding

Let us conclude this section with a silly discussion about encoding. Some compilers allow for the use of UTF in source codes. We do not dispute the advantage of having many more

[12] http://lwn.net/Articles/57135.

[13] Provided there are no blank spaces after the trailing backslashes, which in itself is an interesting observation.

symbols at hand, but we are definitely concerned by the numerous other possibilities offered by UTF-compliant tools.

Consider this valid JAVA code:

```
public class Preprocess {
  public static void ma\u0069n (String[] args) {
    if (false==true)
    { //\u000a\u007d\u007b
      System.out.println("Bad things happen!");
    }
  }
}
```

Code snippet 29.

Despite appearances, from the JAVA compiler's perspective the method name is `main`, the comment mark is immediately followed by a line feed, a closing bracket and an opening one. The `println` command, being out of the `if` block, is therefore executed. That is, using basic escape sequences, the structure of the code is modified.

We have not yet investigated other interesting features such as right-to-left mark which allows for reverse printings (as well as overwrites), but we are definitively worried about them: there is a risk that different tools (such as the editor and the compiler in the previous example) provide different interpretations of the same source, leading to confusion or allowing for obfuscation.

IV. FROM SOURCE CODE TO EXECUTION

Even when the programming language used to develop critical parts of a system offers all the desirable properties, what really matters from the evaluation perspective is the behavior of the program at runtime. Let us explore the long road from source code to execution, and some consequences for those desirable properties.

A. Which program?

Evaluators and interpreters: For critical systems, we have mentioned in Sec. II our concerns regarding the presence of evaluators in a language, as they make program analyses impossible. But we also need to mention that, by definition, such constructs allow for code injection. This applies to languages such as PHP, LISP that have built-in internal evaluators, but also to any other language which provides access to external evaluators.

Web applications are the canonical illustrations, transmitting strings interpreted as SQL queries by database engines. In PHP for example a common mistake is to build the query by concatenating constant strings with user-provided data, e.g. `$query="SELECT * FROM MyTable WHERE id ='".$login."'"`, without realising what would happen should the variable `$login` be assigned value `"' OR 1=1; DROP MyTable; -- "`. But similar concerns exist e.g. when playing with `exec`-like features, using a shell as the interpreter, like `system` or `popen` in C.

There is no definitive conclusion about this type of vulnerabilities. Use of evaluators should be discouraged and submitted to thorough controls. It also justifies appropriate education for the developers, focusing not on recipes but on the principles, which are very generic: they apply not only to web applications in PHP but to any language with mechanisms blurring the frontier between data, metadata and code.

To illustrate the genericity of this notion, let us consider the shell: what would you expect as the result of the command `rm *` in a directory containing a file named `-fr`, for example?

Interpreted language: Various forms of code injections or unexpected executions are discussed in this paper. To palliate such vulnerabilities at system level, there are numerous known mechanisms and established practices. For example, one can control executable files, or rely on enforcing the $W \wedge X$ property to forbid a memory location to be both writable and executable.

But what become such mechanisms with an interpreted language? As the *code* is only read and not executed (from the processor perspective), they offer no protection; the use of interpreted languages therefore requires an update of common security practices and thumb rules. Worse, with Just-In-Time (JIT) compilers, modern interpreted languages need the memory to be writable and executable at runtime, preventing the use of $W \wedge X$ mechanisms, re-opening an avenue for attacks at the native level.

B. Undefined behavior

Discussing side effects, we have illustrated unspecified or inadvisable constructs that can be compiled and executed without warnings. There are other meaningless but supported constructs in C, including very intriguing (and worrying) ones, such as for example pointer arithmetic applied to function pointers[14]. We have also indicated that in the absence of strong typing, some languages try and define the meaning of many possible expressions.

A related problem in some languages is when the compiler leverages the presence of undefined behaviors to optimise the result. For example, in C, since dereferencing a null pointer makes no sense, the compiler can fairly assume that a dereferenced pointer is not null. Thus, in the following code, line 2 results in the assumption that `tun` is not null. The compiler can therefore remove the useless lines 3 and 4 for optimisation:

```
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
if (!tun)
     return POLLERR;
/* use *sk for write operations */
```

Code snippet 30.

This particular code was part of the LINUX kernel in 2009 and led to a vulnerability (CVE-2009-1897). A recent study [WZKSL13] build on similar examples to analyse the interaction between undefined behaviors and compiler optimisations. However, some of these dangerous optimisations do not show in the source code, but are created as a result of previous intermediate steps, which makes it very hard to fix the general issue without drastically impacting the performance of the generated code.

C. Encapsulation and compilation

Many languages offer a form of encapsulation, but in general this abstract characteristic does not survive compilation, producing a single, monolithic memory mapping. This is understandable, as the disappearance of encapsulation has

[14] It works... Well, it compiles, executes, and does *something*.

no observable effect on the execution of the program, whereas performance are as good as possible. Yet this also explain why in dysfunctional situations (such as error injection following a buffer overflow) there is usually no protection nor any detection mechanism.

JAVA, being executed as a bytecode on a JVM, emulate such protections – this is part of the job of the bytecode verifier. Of course, as mentioned in Sec. II, introspection can be a cause of concern, but let us put aside this question for a moment to consider a more intriguing fact.

Indeed, JAVA allows for defining inner classes. Instances of the inner class have direct access to private fields of the containing class, and *vice versa*, as illustrated by this code:

```
public class Innerclass {
  private static int a=42;

  static public class Innerinner {
    private static int b=54;
    public static void print() {
      System.out.println(Innerclass.a);
    }
  }

  public static void main (String[] args) {
    System.out.println(Innerinner.b);
    Innerinner.print();
  }
}
```

Code snippet 31.

Interestingly, inner classes can be defined in JAVA but cannot be represented in bytecode – this is quite remarkable, as it means in theory that the JAVA language is more expressive than the JAVA bytecode and that the former cannot be compiled in the latter.

Yet the previous example can be compiled, so how is it done? The inner class is in fact extracted and compiled independently. In order to maintain accessibility of both the outer and the inner classes to the `a` and `b` fields, the private tags are removed. That is, those fields are now accessible from instances of other classes as well.

In our view, such silent modifications of the code by the compiler should be banned.

D. Memory protections in native binaries

Usually, when the compilation phase produces a native executable, all the compilation units used are grouped in the same memory space, where only coarse grain protection can be enforced: read-only regions and non-executable stacks or data.

As a matter of fact, the link between source code indications and binary memory mapping is sometimes not preserved, but can even be broken. Consider the following C code:

```
int main (void) {
    char* s = "Hello";
    printf ("%s\n",s);
    s[0] = 'h';
    printf ("%s\n",s);
    return 0;
}
```

Code snippet 32.

When compiling it on recent systems, the string literal `"Hello"` is put in the read-only data section at compile-time, leading to a segmentation fault when executing line 4. The problem comes from the compiler that allows for a read-only string to be put into a read-write variable. This inconsistency vanishes when using G++ instead of GCC, or by adding in the options the `-Wwrite-strings` flag, which is off by default due to compatibility issues.

We believe that in some cases, it would be interesting to go further than the current implementations of compilers. Critical parts of a system could benefit from finer-grain memory protections. For example, in object-oriented languages, different classes (or even different instances of the same class) could be compiled as different processes or at least as different threads, relying on operating system mechanisms to provide stronger isolation. This would mean that the compilation steps also consider non-functional properties (like encapsulation) as invariants to be preserved along the way, whereas today they only care to maintain observable behaviors in standard conditions.

E. About serialization

In a static type-checking system, types are pure logical information which have no concrete existence in the actual implementation.

It is a pity because very often types, being so intuitive, are implicitly used in developers' reasonings – that is, some ill-typed situations may never be considered, whereas they can indeed occur. This is for example what make the wrap-and-decipher vulnerability of PKCS#11 [Clu03] so elusive: the attack relies on exporting a cryptographic key which is later re-imported as a message. Keys and messages can indeed be confused in the physical world at execution time, whereas theoretical models can implicitly forbid such a scenario (see [JH09], [Jae10]). To some extent, this also explains why developers may omit to check the consistency of data in complex formats, e.g. trusting the provided uncompressed size field in the header of a compressed file, resulting in buffer overflow vulnerabilities when using C-like languages.

Many modern languages provide an easy way for the developer to store and load binary objects as strings or files, the serialization. This mechanism allows for building complex structures (huge trees or hash tables) only once, saving them on disk and then relying on the deserialization to unfold the object without having to code a parser.

In this context, our remarks about forgotten types apply. In practice, types are related to the interpretation method of such a string, and deserializing data is therefore often an act of faith, relying on the hope that the loaded string will be correctly interpreted as a value of the expected type. In general, modifying a serialized object on disk will lead to a memory error on loading, but the worst case scenario is to forge a serialized object directly pointing at memory cells it should not have access to.

As a matter of fact, the LaFoSec study [ANS13] showed that the OCAML language was affected: during the deserialization, references between values are unfolded, yet no check insures the extracted references correctly point to deserialized values. It is therefore possible to forge a serialized blob containing pointers towards memory cells outside the blob – this is by the way the true meaning of the warning in the OCAML reference manual about `Marshal.from_channel`:

> Anything can happen at run-time if the object in the file does not belong to the given type.

The same is true with the PYTHON `pickle` module, whose documentation[15] explicitly recognises the vulnerability:

> [The] pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

On occasions, type information would be interesting to preserve at runtime as metadata – provided their integrity is ensured. But even when types are enforced at runtime, as it is the case in JAVA bytecode (without integrity protection, but this is not the subject of discussion here), serialization can still lead to serious security problems. Consider the following code:

```
import java.io.*;

class Friend { } // Unlikely to be dangerous!

class Deserial {
  public static void main (String[] args)
    throws FileNotFoundException,
           IOException,
           ClassNotFoundException {
    FileInputStream fis =
      new FileInputStream("friend");
    ObjectInputStream ois =
      new ObjectInputStream(fis);
    Friend f=(Friend)ois.readObject();
    System.out.println("Hello world");
  }
}
```

Code snippet 33.

This code is intended to deserialize an instance of the class `Friend`, but this is not what happen in practice. Indeed, JAVA serialized files contains a reference to their class, and deserializing such a file automatically loads the corresponding class and executes its initialisation code[16], before creating the instance in memory. It is later that this instance is casted – possibly causing an exception, should types not be compatible. But this is far too late if the initialisation code is malicious!

Thus JAVA serialized objects should not be seen as mere data. In 2008, a vulnerability was reported on the way the Calendar class deserialized foreign objects in a privileged context (CVE-2008-5353). In this particular example it was shown that deserializing an object could lead to load a new class and to execute initialisation code with the privileges of the standard library classes.

F. Low-level details matter

To understand how some vulnerabilities operate in order to prevent or patch them, it is often necessary to understand how memory management works in a computer.

Let us start with a format string attack in C[17]:

```
#include <stdio.h>

char *f="%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.\
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n";

void strfmtattack() { printf(f); printf("\n"); }

int main(void) {
  int s=0x40414243;
  int *p=&s;
  strfmtattack();
  if (s!=0x40414243)
    printf("Bad things happen! s=%d\n",s);
  return 0;
}
```

Code snippet 34.

At execution, this code prints various information, ending with `40414243` then `Bad things happen! s=126`. That is, the function `strfmtattack` uses a simple `printf` to read and overwrite variable `s` which is outside its scope and should therefore not be accessible.

Direct stack smashing is also possible in C and may have other interesting consequences, as in the following illustration:

```
#include <stdio.h>
#include <stdlib.h>

void set(int s,int v) { *(&s-s)=v; }

void bad() { printf("Bad things happen!\n"); exit(0); }

int main(void) {
  set(1,(int)bad); printf("Hello world\n"); return 0;
}
```

Code snippet 35.

At execution, this program prints `Bad things happen!` and exits. That is, the `set` function gets a pointer on the stack to overwrite its own return address – modifying execution flow.

Once more, the frontier between code and data appears to be rather thin – here as a consequence of various historical choices such as *Von Neumann*'s architecture and the existence of a single stack mixing data and addresses. This is also possible because of the C language, allowing for low-level manipulations without associated controls. Describing how stack buffer overflows or format string attacks work ([Ale96], [LC02]) requires knowledge on how variables, function arguments and return pointer are actually mapped in the memory. Without this culture, how to imagine that an out-of-bounds write in an array may result in code injection and execution?

We certainly appreciate high-level programming languages and the features they provide for developers, such as automated memory management by a garbage collector preventing many critical bugs. On the other hand, we are not very comfortable with the related idea of teaching only high-level programming languages. Our fear is that if developers no longer need to call `malloc` and `free` because the language they use has a garbage collector, there will be no point of teaching them what the stack and the heap are.

This is plain wrong, as high-level languages *in fine* rely on native binaries and libraries. In particular, it ends up with this type of code[18]:

```
public class Destruction {
    public static void delete (Object object) {
        object = null;
    }
}
```

Code snippet 36.

[15] http://docs.python.org/3/library/pickle.html

[16] That is code not included in any method and marked as `static`.

[17] Recent C compilers emit warnings when compiling this code because of the risk of format string attack.

[18] Source: http://thedailywtf.com/Articles/Java-Destruction.aspx; beyond the source code presented, many comments are worth reading to understand the level of culture regarding memory handling.

By the way, using a garbage collector to handle memory allocations instead of letting the programmer manage memory manually may have security drawbacks.

Indeed, when a program manipulates secrets (passwords or cryptographic keys for instance), one has to minimise the duration of their presence in memory. It also has to ensure secure erasing by overwriting them with other values, so that they are no longer accessible via standard CPU instructions[19]. Those are common practices to tackle e.g. with swaps of memory pages, crashes or other situations in which the content of the memory is dumped and made accessible. Network devices exporting their secrets in a core dump after an easy-to-cause crash exist(ed). *Heartbleed* (CVE-2014-160), the vulnerability affecting OPENSSL, is another recent example of possible sensitive data leakage: a buffer overflow allows an attacker for reading parts of server memory[20]. Even if zeroing data when memory is freed would not have blocked the attack entirely, it would have mitigated some of its consequences.

Yet it is very hard (or even impossible) to control lifetime of a secret data in presence of a garbage collector. For the sake of performances, freed areas are generally not cleared by the garbage collector, for example. Some garbage collectors (e.g. the mark-and-copy flavor) might also spread the secrets several times across the memory.

Note that the same type of concerns may arise from other well-established mechanisms that have been developed to preserve observable behavior of programs but not other types of properties related to security. Erasure by overwritings, for example, can be removed by a compiler during optimisation (as there are no later readings), neutralised by cache mechanisms or flash memory controllers moving around physical writings to avoid fatigue of memory cells, and so on.

V. ABOUT ASSURANCE

> Software and cathedrals are very much the same – first we build them, then we pray
>
> Sam Redwine

Developing correct and robust applications is difficult, and it has become even harder due to the increasing complexity of the manipulated concepts and systems. Furthermore, as mentioned in Sec. I, it is in general useless in industrial environments to have a secure software without knowing for sure that it is indeed secure. This also applies in other domains such as transportation systems, and leads to the notion of certification process, such as [IEC]. In IT security, the equivalent standards are the Common Criteria [CC], which define different evaluation assurance levels (EALs).

Security certification relies on analyses of the program and of its development process by independent evaluators, a process known as security evaluation. Discussing the intrinsic security characteristics of languages therefore requires appropriate consideration of the helps and limits that applies to the evaluation process. This includes for example an appreciation of the genuine understanding of both developers and evaluators of the features and mechanisms used. This is why we do not consider most advanced language constructs as fitted for critical developments, simpler approaches generally allowing for higher level of assurance.

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> C.A.R Hoare

A. About specifications

In our view, developing an adequate level of mastery of the constructs of a language requires both practice and theory. The latter has not to rely on academic publications but can be addressed by proper specifications for the language, e.g. with well-defined semantics for its main constructs, including the standard library.

But consider as a counterexample this extract of the JAVA specification for the `Object` class:

> The *general intent* is that, for any object `x`, the expression: `x.clone() != x` will be true, and that the expression: `x.clone().getClass() == x.getClass()` will be true, but these are *not* absolute requirements. While it is *typically* the case that: `x.clone().equals(x)` will be true, this is *not* an absolute requirement.

In essence, this specification provides no requirements, and it would be unreasonable to expect *anything* about `clone`.

This type of problems arises quite often, and various constructions in several languages appear to be only partially specified, voluntarily or not, explicitly or not. Let us consider for example two extracts of [KR88] for the C language:

> The meaning of "adding 1 to a pointer" and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

As already mentioned, this first extract appears relatively clear, even if it is informal; yet it says nothing about the expected behavior for function pointers.

> The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

This second extract is a good example of a non deterministic specification: at least, the reader is explicitly informed that there is no guarantee about the result e.g. of `-1/2`.

It is worth mentioning at this stage that non-deterministic specifications can be tricky to handle. How would you test a compiler to check that the result of `-1/2` complies with this specification? In general, the proposal is to check that it yields either `0` or `-1`, but we would argue that this is not sufficient. Indeed, an implementation returning one or the other according to other parameters (e.g. time of the day or a secret value)

[19] We are not talking here about physical attacks on memory chips or mass storage systems, so-called *cold boot attacks*, as described in [HSH+09].

[20] To be precise, the data leaked is located in the process heap, and the flaw also affects OPENSSL clients.

would not be rejected. Yet you can be sure that having `-1/2` not always equal to `-1/2` (that is loosing reflexivity) would have consequences on the robustness of any software.

This is an instance of a problem known as the *refinement paradox* in formal methods such as B [Abr96]: given the non-deterministic specification $b \leftarrow \text{getb} \triangleq b := \top \,[]\, b := \bot$ (that is $getb$ is an operation returning a value b which is either true or false), one often consider only the two constant solutions $b \leftarrow \text{getb} \triangleq b := \top$ and $b \leftarrow \text{getb} \triangleq b := \bot$. Yet there are other compliant implementations of getb in which the return value b is still in $\{\bot, \top\}$ but depends upon dynamic values. This notion of *refinement paradox*[21] applies to refinement-based methods such as B, but also to other formal methods such as COQ [Jae10] – and, as illustrated here, in many other, non formal situations.

As a summary, we prefer languages to be specified as completely, explicitly and formally as possible, avoiding non deterministic or partial properties. But we would also like to have tools such as compilers that provide errors and warnings when facing unspecified, discouraged or forbidden constructions.

B. About code signature

Code review by independent evaluators is a standard process in security evaluation.Yet one has to question its relevance e.g. when dealing with object-oriented languages.

Let us review the following JAVA code:

```
class StaticInit {
  public static void main(String[] args) {
    if (Mathf.pi-3.1415<0.0001)
      System.out.println("Hello world");
    else
      System.out.println("Hello strange universe");
  }
}
```

Code snippet 37.

Of course, there is a trick – one that we have already mentionned when discussing deserialization in Sec. IV. It has nothing to do with the value of the constant `Mathf.pi` but with the class `Mathf` itself:

```
class Mathf {
  static double pi=3.1415;
  static { // Do whatever you want here
           System.out.println("Bad things happen!");
           // Do not return to calling class
           System.exit(0); }
}
```

Code snippet 38.

A JAVA program merely accessing `Mathf.pi` will dynamically load the `Mathf` class and execute its initialisation code. The scope of the evaluation is therefore much broader, as non-executable files representing JAVA classes on the execution platform at runtime (rather than on the development platform at compile-time) have to be considered as well.

This is one of the mechanisms supporting *ad hoc* polymorphisms in object-oriented languages such as JAVA. By comparison, the ADA genericity (with code specialisation at compilation) or the OCAML polymorphism (a single code for values of different types) are much more easy to handle.

[21]The term paradox is an overstatement, see e.g. [MM04].

In some cases, to compensate the lack of guarantees on the program which is actually executed, it is possible to rely on signature – provided signature checks cannot be avoided, that the verifier cannot be tampered, that cryptographic keys are adequately protected, that cryptographic protocols ensure required properties, *etc*. It is usual for example in JAVACARD to verify the bytecode and sign the applet off-line, to avoid embedding a bytecode verifier in the card.

Yet defensive checks of source code properties (or proof-carrying codes) and code signatures have very different goals: the former aims at ensuring that a program behaves correctly whereas the latter only deals with its origin. In particular, code signing says nothing about the quality of the code (in terms of correctness or robustness for example) or the competence of the signer.

All in all, code signing is useful for low-level libraries, when other defensive checks are not yet available. Here, the innocuousness of such code should be checked by source code audit and vulnerability analysis, signature only providing authentication on the execution platform (instead of any guarantee about the behavior of the code). In other situation, signature is at best part of a defence-in-depth approach.

C. The critical eye of the evaluator

The different examples presented in the previous sections show that an evaluator should be aware of the numerous traps of programming languages.

When the evaluation process includes a source code audit, a good knowledge of the involved programming languages can help the evaluators, but we believe it is more important that they have a deep understanding of the *features* provided by the language and used by the developers. For example, object-oriented paradigm and serialization mechanisms are pervasive. The potential security issues are the same across programming languages implementing them. We have also discussed the very generic concept of code injection.

As we saw earlier, as the developer's intuition might be shattered by some constructions, it is also important to question the properties advertised by the language and their practical robustness. For example, checking whether or not a private field is accessible may be rewarding, as in the JAVA inner class example discussed in Sec. II.

Finally, as illustrated in the Sec. IV, the evaluation of a product should depend on the intended final build process and the target runtime environment, since low-level details matter, and may vary from one platform to another. It is not rare to observe important differences between development and production builds, like the optimisation level; this may lead to different behaviors in practice, and even to real-world vulnerabilities as in the C undefined behavior case.

D. About formal methods

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> Donald Knuth

At this stage, some of the readers may consider that the previous concerns raised would be solved by using deductive

formal methods, allowing for proving program compliance with specifications, such as e.g. the B method [Abr96] or the COQ proof assistant [Coq].

According to our analyses, this would be overoptimistic, as some of the problems pointed out still apply, and formal methods come with their own traps for inattentive developers or evaluators. This is discussed at length in [JH09], [Jae10], dealing with validity and completeness of the specification, the limits of expressivity of the formal languages, inconsistencies in the formal theory or bugs in the tools, trusted translations from formal to standard (executable or compilable) languages, and last but not least not satisfying explicit or implicit hypotheses when using proven software.

For example, in 2004 a variant of SSH was proven secure [Bel04], whereas in 2009 a plaintext-recovering attack against this very variant was discovered [APW09]. In the former paper, Bellare et al. made an implicit assumption by working with binary streams already split into messages, whereas the attack presented indeed relied on this split operation.

VI. LESSONS LEARNED AND PROPOSALS

We have considered numerous aspects of programming languages, trying to identify concerns when security is at stake. Rather than to attempt to provide definitive conclusions about which language should be used, we just provide a list with a few lessons that we learned during our journey, as well as some proposal for the way ahead.

A. Languages and tools design

First of all, as mentioned in the very beginning of this paper, we have been surprised to find that security of languages, when considered at all, was often so with a very narrow scope or wearing blinkers[22]. It is already common knowledge that designing languages is *hard*, in all likelihood an order of magnitude harder than learning and mastering a language. However, in the light of our work, some additional requirements should be taken into account.

If we certainly appreciate works related to more expressive type systems or the native integration of security mechanisms, we would like to avoid having feet of clay. This is why we advocate additional foundation works, defining which characteristics of languages are desirable to cope with security for critical developments.

This requires considering common vulnerabilities or limits, as well as studies of how properties can be ensured or preserved from the theory to the real execution environment, and careful consideration of the robustness of the proposed mechanisms. We do not expect (nor request) new languages to be defined, but at least that evolutions of existing ones are discussed with some considerations about security in mind.

To add a few remarks about languages design when security is at stake:

- language principles, whatever their theoretical elegance, must never go against developers' (or evaluators') intuition;
- eliminate non-determinism in the specification of a language, and make explicit as far as possible undefined behaviors (if any);
- always balance the advantages of new constructions with the added complexity for developers to master their semantics.

Beyond the language design itself, some attention should be paid to the associated tools, such as for example the compilers, debuggers, analysers or runtimes:

- avoid laxism, and track unspecified or undefined constructs to signal them through warnings or errors;
- ban silent manipulations;
- consider additional security options, e.g. to disable some optimisations or to enable non functional features (such as effective erasure);
- consider new compilation invariants or instrumentation, e.g. to preserve type information, encapsulation, execution flow (for example to cope with physical or logical fault injections);
- ease inspection of actual program executed at runtime and traceability with source code for evaluation;
- the tools should protect themselves against specifically crafted malicious inputs;
- long-term goals should include the development of trusted (or certified) tools, such as the CompCert initiative [Ler11], [Dar09].

The robustness principle (also known as *Postel*'s law) states that you should be conservative in what you do and liberal in what you accept. It aims at improving interoperability, but is not, in our view, appropriate when security is at stake.

B. Training developers and evaluators

Quite often IT security is expected to be managed only and independently by IT security experts through patches, intrusion detection systems, boundary protection devices, and so one. Yet we consider it is impossible to deal appropriately with security this way. Every developer, for example, should be security-literate enough in order not to introduce vulnerabilities to start with.

Therefore, beyond messages addressed to language and tool specialists, we would like to mention a few recommendations related to the education of developers (at least those dealing with critical systems and security concerns), that are also applicable for the education of evaluators.

Developers should also be ready, beyond functional approaches (checking that what should work works), to also adopt dual approaches (checking that what should not happen never happens). This includes for example worst-case reasonings related to unsatisfied pre-conditions, out-of-range values, ill-formed messages, *etc*. and can be supported by review of most common vulnerabilities and attacks.

[22] JAVA 8, published in March 2014, advertises on *improved security*, that is new cryptographic algorithms, better random generators, support for TLS, or PKCS♯11 to cite a few. Yet we worry about the actual gains security-wise as these evolutions, in practice, may translate into additional vulnerabilities in an ever-more complex stack of codes and protocols built on sand.

Last but not least, as attackers are always looking for the weakest link, developers should be able to have a broad vision encompassing most of the avenues of attacks. To this aim, we have the feeling that one has to learn the basics in several domains such as language semantics, compilation theory, operating system principles, computer architecture. These are, basically, the subjects that have been discussed in this paper when considering illustrations of our concerns.

VII. Conclusion

In this paper, we presented some of the points we consider worth looking at while assessing the contribution of a language to the security of the programs written with this language, be they at the theoretical or the syntactic level, be they related to the compilation, the execution or the evaluation phase. This led us to propose some recommendations. We strongly believe that all the presented aspects should be taken into account when considering what languages bring to the security table.

References

[Abr96] J.R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Ale96] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 49, 1996.

[ANS10] ANSSI. Sécurité et langage Java. http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/securite-et-langage-java.html, 2010.

[ANS13] ANSSI. LaFoSec : Sécurité et langages fonctionnels. http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/lafosec-securite-et-langages-fonctionnels.html, 2013.

[APW09] M.R. Albrecht, K.G. Paterson, and G.J. Watson. Plaintext recovery attacks against SSH. In *IEEE SSP*, 2009.

[Atw] J. Atwood. Coding Horror. http://www.codinghorror.com/blog.

[Bel04] M. Bellare. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM TISS*, 7:206–241, 2004.

[CC] ISO/IEC 15408: Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org.

[CJRR99] S. Chari, C.S. Jutla, J.R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, pages 398–412, 1999.

[Clu03] J. Clulow. On the security of PKCS#11. In *CHES*, 2003.

[Coq] The Coq proof assistant. http://coq.inria.fr.

[Dar09] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7, 2009.

[DW] The daily WTF: Curious perversions in information technology. http://thedailywtf.com.

[FO] functional orbitz. http://functional-orbitz.blogspot.fr.

[GP99] L. Goubin and J. Patarin. DES and Differential Power Analysis The "Duplication" Method. In *CHES*, 1999.

[HA05] K. Hayati and M. Abadi. Language-based enforcement of privacy policies. In *Privacy Enhancing Technologies*, LNCS 3424. Springer, 2005.

[HSH+09] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[IEC] IEC 61508: Functional safety of electrical, electronic, programmable electronic safety-related systems. http://www.iec.ch/zone/fsafety/.

[Jae10] É. Jaeger. *Study of the Benefits of Using Deductive Formal Methods for Secure Developments*. PhD thesis, EDITE, 2010.

[JH09] É. Jaeger and T. Hardin. A few remarks about formal development of secure systems. *CoRR*, abs/0902.3861, 2009.

[Koi] S. Koivu. (Slightly) Random Broken Thoughts. http://slightlyrandombrokenthoughts.blogspot.fr.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

[LC02] K. Lhee and S.J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33:423–460, 2002.

[Ler11] X. Leroy. Verified squared: does critical software deserve verified tools? In *38th symposium Principles of Programming Languages*. ACM Press, 2011. Abstract of invited lecture.

[Mil84] R. Milner. A proposal for standard ml. In *LISP and Functional Programming*, pages 184–197, 1984.

[MM04] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer Verlag, 2004.

[MT91] R. Milner and M. Tofte. *Commentary on standard ML*. MIT Press, 1991.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[WL99] P. Weis and X. Leroy. *Le langage Caml*. Dunod, 1999.

[WZKSL13] X. Wang, N. Zeldovich, M.F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.