Jasa : The new Jasmin Safety Analyser based on MOPSA

Juguet Léo

March - Jully 2024

Abstract

We present a new safety checker for Jasmin, a programming language for high-assurance and high-speed cryptographic code implementation. The safety checker detects initialization of scalars, initialization of arrays to a certain extent, and access out of bounds without unrolling loops using widening. This new safety checker is mostly based on the MOPSA library. It is undoubtedly faster than the previous one, thanks to the support of function contract and to do modular analysis. This new safety checker, created with MOPSA, also enables checking more properties than previous safety checker, with the ability to analyze the values.

1. Introduction

Jasmin [1] is a programming language that aims to be secure, particularly for cryptographic code. Its compiler is written in Coq and provides a proof that the code will be correctly translated to assembly, under certain assumptions. These assumptions are checked by a safety checker, but the previous one was too slow, and not as precise as we wanted. Moreover the safety checker is hard to maintain today, and doesn't allow modular analysis. The goal of this internship was to create a new, more efficient, more maintanable, and more precise safety analyzer that would be able to check that the conditions on any Jasmin code is verified.

For this task, I used MOPSA, a static analyzer library that aims to be modular, in order to easily add a frontend for the Jasmin language. By relying on the third-party library MOPSA, which maintains the backend of the abstract interpretation, the resulting codebase is more maintainable.

2. Context

Writing a secure program is hard, as there are many considerations to take into account, and often small mistakes are involuntarily made by programmers, such as badly definining variables or accessing out-of-bound memory. These mistakes can lead to writing at unsafe locations, leading to an information leak. Hence, a tool is needed to detect mistakes.

In Jasmin, a safety analyzer already exists, but there is a main problem: The current implementation is too slow for big programs, it's impossible to run an analysis on an independent function without inlining other functions. Moreover the safety checker also asks to maintain a backend to be able to do abstract interpretation.

The main goal of this internship was to explore MOPSA to see if it could be used to replace the old safety checker.

2.1. Contribution

During this internship, I wrote a brand new safety analyzer for Jasmin [1] using the MOPSA [5] library. This safety checker is modular, meaning that functions can be analyzed independently. It can check if scalars are properly initialized, as well as arrays and memory, and ensure there are no out-of-bounds accesses.

The safety checker uses contracts to check properties and offers a modular analyzer. It is also faster than the previous one.

Contents

1. Introduction 1
2. Context 1
  2.1. Contribution 1
3. Jasmin 2
  3.1. Safety 3
4. Overview of Abstract Interpretation 3
  4.1. Widening 5
5. Overview of MOPSA 5
  5.1. Semantic of statements 6
    5.1.1. Semantic of Jasmin 7
    5.1.2. Semantic of Jasmin Abstraction 7
6. Initialisation of scalar 8
7. Initialisation of arrays 9
  7.1. The difficulty 9
  7.2. Aproaches 9
  7.3. 3 Segments 9
    7.3.1. Representation 10
    7.3.2. Setter 10
    7.3.3. Access 11
    7.3.4. Example 11
    7.3.5. Concretization 12
8. Contract and function call 12
9. Memory 12
10. Performance and Implementation 13
11. Conclusion and future work 14
12. Acknowledgements 15
13. Meta-Information 15
Bibliography 15

3. Jasmin

Jasmin is a programming language for writing high-assurance and high-speed cryptography code. The compiler is mostly written and verified in Coq, and except for the parser, the other code written in OCaml is also verified in Coq [1]. Jasmin has some tools to translate a Jasmin program to EasyCrypt or CryptoLine to allow developers to prove that their code is correct. However, this translation and the compilation are correct only if the safety properties are verified. The safety properties are checked by a safety checker.

Jasmin has a low-level approach, with a syntax that is a mix between assembly, C, and Rust. Jasmin has 3 types:

- bool for boolean
- inline integers that are not saved somewhere but directly written in the source code when the compiler unrolls the loop
- words of size with the size in $\{8, 16, 32, 64, 128, 256\}$
- arrays of words of fixed length at compile time
- reg variables that can hold addresses

The user has to specify if a variable has to be in a register or on the stack. But this will not affect the safety analysis.

A Jasmin program is a collection of:

- parameters (that are removed right after the parsing)
- global variables (that are immutable)
- functions (they can't be recursive)

The control flow of a Jasmin program is simple, with blocks, if statements, for loops, and while loops, all of which have a classic semantic of an imperative language.

3.1. Safety

In Jasmin, safety is formally defined as "having a well-defined semantics", as specified in Coq. A function is deemed safe if it can reach a final memory state and result values such that executing the function from the initial state successfully terminates in that final state. The properties that must be verified include the following:

Definition 1: Safety properties
The safety properties of a Jasmin program are :
- all scalar arguments are well initialized
- all return scalars are well initialized
- there is no out-of-bounds access for arrays
- there is no division by zero
- termination
- alignment of memory access

Out-of-bounds access is a common condition in software verification, as it helps prevent writing to unauthorized memory locations and avoids information leaks. Checking for division by zero is also crucial to ensure that the program does not crash. Memory access alignment is required by certain architectures and can enhance execution speed.

Termination is important, especially in cryptographic programs, as most of them need to execute in constant time. Scalars must be initialized to ensure that they are properly represented in the generated assembly code.

In the following report, we will not talk about division by zero, even if a prototype of division by zero detection is implemented, because Jasmin is focused on cryptography code, and in particular constant-time programs. Division operations are not used because they are not constant-time operator. We will focus on the initialization of scalars and arrays, as well as out-of-bounds access to arrays

4. Overview of Abstract Interpretation

> **Note** : this part and Section 5 are mostly a summary of [5]. For proof and more details, please refer to this document.

Abstract interpretation is a technique in the field of static analysis of programs. It analyzes a program by an upper approximation way, without having to execute the program.

Abstract interpretation was formalized by Patrick Cousot and Radhia Cousot [2]. All concrete values are replaced by an abstract value, such that all possible values in a normal execution are included in the concretization of the abstract value.

A perfect abstract interpretation would calculate the exact set of possible states of a program during or at the end of the interpretation. However, it is generally impossible to calculate such a set.

Therefore, we compute an approximation. To do this, we define a poset over the set of possible states of a program, and we say that the abstraction is sound, meaning it is a correct approximation, if it is possible to derive an upper set of possible states of the program from the abstraction.

Definition 2: Poset
A poset or partial order set is a couple $(X, \sqsubset)$ with X a set and $\sqsubset \in X \times X$ a relation reflexive, anti-symmetric and transitive

Definition 3: Sound abstraction
Let A be a set $(C, \subseteq)$ a poset, and $\gamma \in A \to C$ the concretisation function. a is a sound abstraction of c if $c \subseteq \gamma(a)$

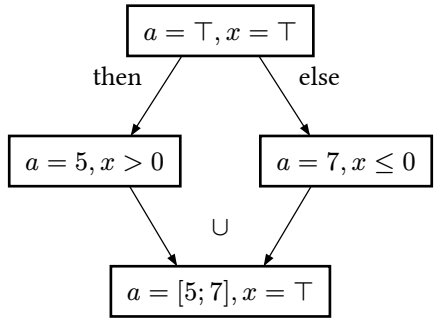
Intuitively, the abstraction calculates an overbound of the real possible values. The abstraction is correct even if it detects a bug that didn't exist, but it is incorrect if it fails to detect a bug that does exist.

The notion of sound abstraction can also be extends to function.

Definition 4: Sound abstraction operator
Let A be a set $(C, \subseteq)$ a poset, and $\gamma \in A \to C$ the concretisation function. $f^{\#} \in A \to A$ is a sound abstraction of $f \in C \to C$ if $\forall a \in A, f(\gamma(a)) \subseteq \gamma\big(f^{\#}(a)\big)$

```
fn f(reg u64 x)
{
  reg u64 a;

  if x > 0 {
    a = 5;
  }else{
    a = 7;
  }
}
```



Listing 1: Example of abstraction of a program

For example in Listing 1, first we abstract a and x by $\top$ value, that represent the full range of possible values, when we execute the `then` branch we have the approximation that $x > 0$ and that $a = 5$, but in the `else` branch we have that $x \leq 0$ and $a = 7$. At the end we join the two branch, to limit the number of different abstract state that we have, and the approximation give tha a is in $[5; 7]$ and x can take all values. This is an overapproximation because $a = 6$ is impossible with the code of the example.

For example, in Listing 1, we first abstract a and x with the $\top$ value, which represents the full range of possible values. When we execute the `then` branch, we have the approximation that $x > 0$ and $a = 5$. However, in the `else` branch, we have $x \leq 0$ and $a = 7$.

At the end, we join the two branches to limit the number of different abstract states we have, and the approximation gives that a is in the range $[5; 7]$ while x can take all values. This is an overapproximation of the possible states, because $a = 6$ is impossible with the code in the example.

4.1. Widening

The widening operator was initially introduced in a Cousot's Cousot's paper [2]. The widening operator was defined to find a fixpoint in the approximation of while loops, like it's impossible to compute the fixpoint of all loops with the guaratee that this calcul terminate. The semantic of while loop is given in Section 5.1.

Definition 5: widening
A binary operator ∇ is defined by
- $\nabla : A \times A \to A$
- $\forall (C, C') \in A^2, C \leq C \nabla C', C' \leq C \nabla C'$
- $\forall (y_i)_{i \in \mathbb{N}} \in A$, the sequence $(x_i)_{i \in \mathbb{N}}$ computed as $x_0 = y_0, x_{i+1} = x_i \nabla y_{i+1}, \exists k \geq 0 : x_{k+1} = x_k$

The 3rd condition of the widening operator is important to guarantee that the analysis will finish, but this does not permit us to conclude that the loop can finish, unless the index used in the loop is bounded.

5. Overview of MOPSA

MOPSA, which stands for Modular Open Platform for Static Analysis, is a research project that develops a tool in OCaml of the same name. The goal is to create static analyzers based on abstract domains modularity.

MOPSA work with Domain, named composable abstract domain in [5]. These domains handle an abstract representation of the things they want to abstract (e.g. an interval of values, or the initialization states). These domains also take partial expression and statement transfer functions, which indicate how certain instructions modify the abstract representation (e.g. `a = a + 2` has the effect of adding 2 to the interval representation of `a`).

Definition 6: Composable abstract domain
A value abstract domain is :
- an abstract poset $\left(D^{\#}, \sqsubseteq^{\#}\right)$
- a smallest element $\bot$ and a largest element $\top$
- Sound abstractions of set union and intersection $\bigsqcup^{\#}, \bigsqcap^{\#}$
- a widening operator ∇
- a partial expression and statement transfer functions, operating on the global abstraction state $\Sigma^{\#}$. $\mathbb{E}^{\#}_{D^{\#}}[\![\text{expr} \in \text{E}]\!], \mathbb{S}^{\#}_{D^{\#}}[\![\text{stmt} \in S]\!]$
- Concrete input and output states of the abstract domain, written D^{in} and D^{out}
- a concretisation operator $\gamma : D^{\#} \to P(P(D^{\text{in}}) \times P(D^{\text{out}}))$

In a configuration file, the programmer defines how different domains coexist. When composing a domain, the system takes the first domain that can handle a given expression or statement.

The Figure 2 illustrates a simplified configuration. The red domains are defined specifically for Jasmin. The blue domains are defined by Mopsa for his universal language. Some domains only translate Jasmin statements into equivalent statements in the universal language offered by Mopsa, like the loop domain that translates Jasmin's while and for loops into a while loop of the universal language.

Other domains, like array initialization, provide a special abstraction for Jasmin, in this case to handle the initialization of arrays (Section 7). In this simple example, the system first tries to see if the intraproc domain from the Jasmin frontend can handle the current expression or statement. If so, that domain will handle the case. Otherwise, the system checks the following domains.

Some cases will try to execute two different domains, like array out-of-bounds and array initialization, where one domain checks for out-of-bounds access and the other checks and modifies the abstraction to determine if an array is properly initialized. A domain can call other domains. For example, if we have `a = b[i]`, the domains that handle integer assignments will ask for the value of the expression `b[i]`, and the same mechanism described before will be used to find the right domains (`array init` to have the value and `array out of bounds` to check that there is no error).

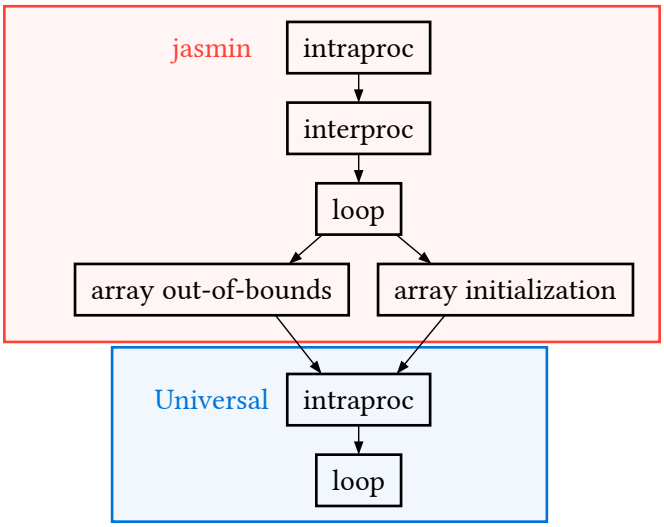


Figure 2: Simplified domain configuration

This way of handling abstraction offers a simple method for developers to add abstractions. The developer only needs to extend the AST of MOPSA to translate their language into the MOPSA AST, which is straightforward with OCaml's extensible types. After that, the developer simply defines new domains to handle the new AST cases they have added. Moreover, MOPSA also provides the possibility to use a reduced domain to avoid requiring the developer to redo things that are already well-defined, such as the `Value Abstract Domain`, which allows for not having to reimplement a full domain. This Value Abstract Domain was used to analyze the initialization of scalar variables (Section 6).

Definition 7: Value abstract domain
A value abstract domain is :
- a poset $\left(D^{\#}, \sqsubseteq^{\#}\right)$
- a smallest element $\bot$ and a largest element $\top$
- a sound abstraction of :
  - constant and intervals
  - binary operators
  - set union and intersection
- a widening operator ∇
- a concretisation operator $\gamma : D^{\#} \to P(\mathbb{Z})$

5.1. Semantic of statements

5.1.1. Semantic of Jasmin

We write $\mathbb{E}[\![e]\!] : S \to P(\mathbb{Z})$ the semantic of an expression e. Variables are evaluate in a context σ.

$$\mathbb{E}[\![v]\!]\sigma = \{\sigma(v)\}$$
$$\mathbb{E}[\![z \in \mathbb{Z}]\!]\sigma = \{z\}$$
$$\mathbb{E}[\![e_1 \divideontimes e_2]\!] = \mathbb{E}[\![e_1]\!] \divideontimes \mathbb{E}[\![e_2]\!] \text{ with } \divideontimes \in \{+, -, \times, \div, \%\}$$

With $A \divideontimes B = \{x \divideontimes y \mid x \in A, y \in B\}$ for $\divideontimes \in \{+, -, \times\}$ and $A \divideontimes B = \{x \divideontimes y \mid x \in A, y \in B, y \neq 0\}$ for $\divideontimes \in \{\div, \%\}$

We write $\mathbb{S}[\![s]\!] : S \to P(S)$ the semantic of an expression s. And we defined a conditional operator $\mathbb{C}[\![c]\!] : P(S) \to P(S)$, that filter the state that can statify the condition c.

$$\mathbb{C}[\![e_1 \square e_2]\!]\Sigma = \{\sigma \in \Sigma \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\sigma, \exists v_2 \in \mathbb{E}[\![e_2]\!]\sigma, v_1 \square v_2\}$$

$$\mathbb{S}[\![s_1; ...; s_n]\!] = \mathbb{S}[\![s_n]\!] \circ ... \circ \mathbb{S}[\![s_1]\!]$$
$$\mathbb{S}[\![\text{if } c\{s_t\} \text{ else } \{s_f\}]\!] = \mathbb{S}[\![s_t]\!] \circ \mathbb{C}[\![c]\!] \cup \mathbb{S}[\![s_f]\!] \circ \mathbb{C}[\![\neg c]\!]$$
$$\mathbb{S}[\![x = e]\!]\Sigma = \{\sigma[x \leftarrow v] \mid \sigma \in \Sigma, v \in \mathbb{E}[\![e]\!]\sigma\}$$
$$\mathbb{S}[\![\text{for } v = c_1 \text{ to } c_2\{s\}]\!] = \mathbb{S}[\![v = c_1; \text{while } v < s\{s; v = v + 1;\}]\!]$$
$$\mathbb{S}[\![\text{while } c\{s_1\}(s_2)]\!] = \mathbb{S}[\![s_2; \text{while } c\{s_1; s_2\}]\!]$$
$$\mathbb{S}[\![\text{while } c\{s_1\}]\!]\Sigma = \mathbb{C}[\![\neg c]\!] \left(\bigcup_{n \in \mathbb{N}} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^n \Sigma \right)$$

In Jasmin, a function only have one `return` statement, at the end of the function.

5.1.2. Semantic of Jasmin Abstraction

Because it's not possible to calculate the exact semantic of jasmin, we calculate an overapproximation of states. In particular for while loops, we didn't try to calculate the fixpoint of each possible iteration, but we use the widening operator (defined in Section 4.1), we add a # in exponent to mark that the calcul is in the abstraction.

$$\mathbb{S}^\#[\![s_1; ...; s_n]\!] = \mathbb{S}^\#[\![s_n]\!] \circ ... \circ \mathbb{S}^\#[\![s_1]\!]$$
$$\mathbb{S}^\#[\![\text{if } c\{s_t\} \text{ else } \{s_f\}]\!] = \mathbb{S}^\#[\![s_t]\!] \circ \mathbb{C}^\#[\![c]\!] \cup \mathbb{S}^\#[\![s_f]\!] \circ \mathbb{C}^\#[\![\neg c]\!]$$
$$\mathbb{S}^\#[\![x = e]\!]\Sigma = \{\sigma[x \leftarrow v] \mid \sigma \in \Sigma, v \in \mathbb{E}^\#[\![e]\!]\sigma\}$$
$$\mathbb{S}^\#[\![\text{for } v = c_1 \text{ to } c_2\{s\}]\!] = \mathbb{S}^\#[\![v = c_1; \text{while } v < s\{s; v = v + 1;\}]\!]$$
$$\mathbb{S}^\#[\![\text{while } c\{s_1\}(s_2)]\!] = \mathbb{S}^\#[\![s_2; \text{while } c\{s_1; s_2\}]\!]$$
$$\mathbb{S}^\#[\![\text{while } c\{s_1\}]\!]\sigma^\# = \mathbb{C}^\#[\![\neg c]\!] \lim \delta^n(\bot) \text{ with } \delta(x^\#) = x^\# \nabla (\sigma^\# \sqcup^\# \mathbb{S}^\#[\![s]\!] \circ \mathbb{C}^\#[\![c]\!]\ x^\#)$$

To have a better approximation of loops, Mopsa, always unrolls the first iteration of the loop. For loops, if the user has the guarantee that the loops terminate, it is also possible to unroll the loop to have a better approximation. However, this slows down the analysis, because each iteration of the loop is simulated and there is no guarantee that the analysis will terminate. The user can also choose to unroll only a fixed number of iterations, and then perform the widening operation afterwards. This also provides a better approximation, but it slows down the analysis.

The semantics of these statements and instructions are classic in abstract interpretation, so Mopsa already has domains for its universal language that can be reused. The Jasmin expressions and

statements can be simply translated to the universal language, and this translation is natural without any surprises.

6. Initialisation of scalar

In Jasmin, all scalar arguments or return values of a function have to be initialized. Moreover, a variable is well-initialized if it is assigned to a well-initialized expression. A well-initialized expression is defined according to the Figure 3.

$$\mathbb{E}[\![a \star b]\!] = \begin{cases} \text{if } \mathbb{E}[\![a]\!] = \text{INIT} \wedge \mathbb{E}[\![b]\!] = \text{INIT} \text{ then INIT} \\ \text{else NOT\_INIT} \end{cases} \text{whith} \star \in \{+, -, *, /, \%\}$$

$$\mathbb{E}[\![\bullet\, a]\!] = \mathbb{E}[\![a]\!] \text{ whith} \bullet \in \{+, -\}$$

$$\mathbb{E}[\![c]\!] = \text{INIT with } c \text{ a constant}$$

$$\mathbb{S}[\![v = e]\!]\Sigma = \{\sigma\{v \leftarrow i\} \mid \sigma \in \Sigma, i \in \mathbb{E}[\![e]\!]\sigma\}$$

Figure 3: concrete semantic of variable initialization

We defined a slighty modify value abstract domain, where we add a MAYBE_INIT value, to be able to have a more precise abstraction.

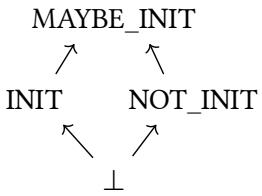


Figure 4: Poset of Init domain

The $\bot$ value of this abstraction is never reached, and is only present to have a lattice. MAYBE_INIT play the role of $\top$.

$$a \cap^{\#} b = a \nabla^{\#} b = a \cup^{\#} b = \begin{cases} a \text{ if } a = b \\ \text{MAYBE\_INIT if } a \neq b \end{cases}$$

$$\mathbb{E}^{\#}_{\text{Init}}[\![a \star b]\!] = \mathbb{E}^{\#}_{\text{Init}}[\![a]\!] \cup^{\#} \mathbb{E}^{\#}_{\text{Init}}[\![b]\!] \text{ whith} \star \in \{+, -, *, /, \%\}$$

$$\mathbb{E}^{\#}_{\text{Init}}[\![\bullet\, a]\!] = \mathbb{E}^{\#}_{\text{Init}}[\![a]\!] \text{ whith} \bullet \in \{+, -\}$$

$$\mathbb{E}^{\#}_{\text{Init}}[\![c]\!] = \text{INIT with } c \text{ a constant}$$

Figure 5: Abstract semantic of variable initialization

We also add a special instruction `InitVariable(var)` that takes a variable and return the same context where the variable in argument is now initialized.

Inside abstract interpretation these values can be interpret has :

- MAYBE_INIT there exist a path were the variable can be initialized
- INIT for all path the variable is initialized
- NOT_INIT for all path the variable is not initialized

We also defined the concretization function to numerical value $\gamma_{\text{Init} \to \text{num}}$ such that $\gamma_{\text{Init} \to \text{num}}(v) = \left\{\top_{\text{typeof}(v)}\right\}$ and to `Init` concrete domain $\gamma_{\text{Init} \to \text{concrete init}}(v) = \begin{cases} \{\text{INIT}\} \text{ if } v = \text{INIT} \\ \{\text{NOT\_INIT}\} \text{ if } v = \text{NOT\_INIT} \\ \{\text{INIT, NOT\_INIT}\} \text{ if } v = \text{MAYBE\_INIT} \end{cases}$.

This is a value abstract domain. So by the property 2.5 of [5], MOPSA built a non-relationnal abstract semantics, that is a sound abstraction of the concrete semantics.

7. Initialisation of arrays

Determining which parts of an array are initialized is essential for validating functions that access arrays

7.1. The difficulty

The main difficulty in proving that an array is well-initialized lies in ensuring that the over-approximation of the index used to set a cell in a loop does not lead to a situation where it falsely appears that the entire array is initialized. For example, in Listing 2, the over-approximation might suggest that `a[x] = 5` for $x \in [2; 512]$, while in reality, only the even indexes are initialized

```
inline int i;
stack u32 a[512];
for i = 0 to 256 {
  a[2*i] = 5;
}
```

Listing 2: partial initialisation of an array

7.2. Aproaches

To initialize an array, different methods exist:

Expanding the array: A first approach is to define a variable for each cell of an array. This can work in Jasmin because arrays have a fixed size. However, this will be memory-consuming, and there is a risk of slowing down the analysis, because the analysis has to iterate over all cells when we do an assignment with an index that has an overapproximation value in the analysis, and this ask to unroll loops.

Array smashing: This approach abstracts a variable by a single variable that represents the entire content of the array, but this will not work in your case to prove the initialization.

Parametric segmentation functor: The approach described in the Cousot's Cousot's paper [3,4] defines a functor that takes 3 different domains to represent values and bounds. The advantage of this algorithm is that it can show that an array is initialized even if it is not initialized from one border to the center. However, the problem with the implementation described is that the functor takes three domains, and the function deals with values in that domain, particularly with some symbolic equalities on the bounds. This way of dealing with domains is not really in the spirit of MOPSA, which prefers to deal with domains and have less information about them. The symbolic equalities would require reimplementing the relational-domain already available in MOPSA, in order to be able to do symbolic equalities of a variable in two different contexts. However, this would ask us to modify MOPSA itself. We did try to do this, but we finally chose a simpler implementation, for reasons of time and because the simpler algorithm covers a sufficiently large number of Jasmin programs.

7.3. 3 Segments

We want deal with to constraints. First we want avoid to have one variable for each cells of the arrays. Because create a variable for each cells, means that each cells has a representation in each abstract domain, like we don't know who many domains we have the memory needed to represent a simple array can grow rapidly. Moreover this also means that we need to modify each variable and check each variable, this can also slow down our safety checker. Secondly, we want to be able to prove that an array is well initialised without unrolling loop, because unrolling loop slowdown the analysis, but give a more precise analysis for loop with invariant difficult to infer, see Section 10.

So we finally move to a more simpler algorithm that can only prove the initialization of arrays if we initialize from border of the array to the center. In pratices, this is the case in a majority of cases that jasmin deal. This initialization is a form of array partitionning but with a fixed number of partition at 3.

7.3.1. Representation

We suppose that we have a numerical domain, if possible a relational domain, $D^{\#}_{\text{num}}$ and a domain that represents initialization $D^{\#}_{\text{init}}$ (a numerical domain can also be used to analyze the values of the array) in the configuration of the analyzer. We represent each arrays with 3 segments, so only 7 variables are needed to represent an array. The bounds of segments are represented by variables handled by the $D^{\#}_{\text{num}}$ domain, while the values of the segments are represented in the $D^{\#}_{\text{init}}$ domain. Four variables are to represent the limit of segments, and three are for representing the value in the segment. So an array a can be represented like that $a = \{b_0\}s_1\{b_1\}s_2\{b_2\}s_3\{b_3\}$ with the properties $0 \leq b_1 \leq b_2 \leq \text{len}(a)$. Moreover b_0 and b_3 are always constant with repectively the value 0 and $\text{len}(a)$.

Variables never change, whe only reassign variable with new values, this permit to delegate operations, union, intersection, to other domain were the variable live.

Initially we have :

$$b_0 = 0, b_1 = 0, b_2 = b_3, b_3 = \text{len}(a), s_1 = s_2 = s_3 = \top \text{ and not init}$$

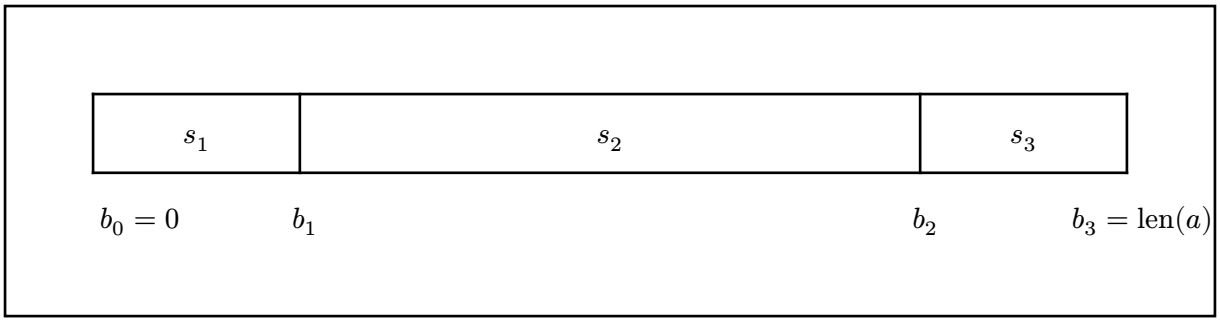


Figure 6: Representation of an array

This method is pretty simple to implement in mopsa. The only case that you need to deal is when you want get a result or when we want assign a value.

7.3.2. Setter

The intuition: The representation with three segments has two “movable” bounds, b_1 and b_2. In many cases, the center segment is not initialized, meaning that the extremities contain more precise information. To avoid losing this information, we extend the extremity segments toward the middle only when we are certain that the bounds will always be included in the defined interval.

Let suppose that we do $a[i] = e$ with i and e two expression. First we suppose that $\gamma(\mathbb{E}[\![i]\!]) \subseteq [0; \text{len}(a)]$, if this is not the cases, we raise an out of bound exception and we continue with this assumption.

We have the following semantics :

$$\mathbb{S}^\#[\![a[i] = e;]\!] = \bigcup \begin{cases} \mathbb{S}^\#[\![b_1 = i + \text{len}; s_1 = \begin{cases} e \text{ if } i{=}0 \\ s_1 \cup^\# e \end{cases}]\!] \circ \mathbb{C}^\#[\![b_1 = i \wedge i + \text{len} \leq b_2]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![b_1 = i + \text{len}; b_2 = i + \text{len}; s_1 = \begin{cases} e \text{ if } i{=}0 \\ s_1 \cup^\# e \end{cases}]\!] \circ \mathbb{C}^\#[\![b_1 = i \wedge i + \text{len} > b_2]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![b_1 = i; b_2 = i + \text{len}; s_2 = e]\!] \circ \mathbb{C}^\#[\![b_1 = i + \text{len} \wedge i + \text{len} = b_2]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![s_2 = s_2 \cup^\# e]\!] \circ \mathbb{C}^\#[\![b_1 < i \wedge i + \text{len} < b_2]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![b_2 = i; s_3 = \begin{cases} e \text{ if } i + \text{len} = \text{len}(a) \\ s_3 \cup^\# e \end{cases}]\!] \circ \mathbb{C}^\#[\![b_1 < i \wedge b_2 = i + \text{len}]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![s_3 = s_3 \cup^\# e]\!] \circ \mathbb{C}^\#[\![b_1 < i \wedge b_2 < i + \text{len} \wedge b_2 < i]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![b_2 = i; s_3 = s_3 \cup^\# e]\!] \circ \mathbb{C}^\#[\![b_1 < i \wedge b_2 < i + \text{len} \wedge b_2 \geq i]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![s_1 = s_1 \cup^\# e]\!] \circ \mathbb{C}^\#[\![i < b_1 \wedge i + \text{len} < b_1]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \\ \mathbb{S}^\#[\![b_1 = i + \text{len}; s_1 = s_1 \cup^\# e]\!] \circ \mathbb{C}^\#[\![i < b_1 \wedge i + \text{len} \geq b_1]\!] \circ \mathbb{C}^\#[\![i \in [0; \text{len}(a)]]\!] \end{cases}$$

In practice, we use a $\cup^\#$ operation that joins the possible values of two expressions. This is effectively a convex join in the interval domain. It's essentially a case disjunction to determine if we can extend the segment on the left or right in the direction of the middle.

When using the numeric relational domain of MOPSA, in the majority of cases, if we initialize from left to right, the first case is the one that is verified, even with the widening of bounds and the index in loops.

This simpler approach, while potentially less precise than a full symbolic equality implementation, has proven sufficient for a large number of Jasmin programs, allowing us to deliver a working analysis within the given time and resource constraints.

7.3.3. Access

To access an element of a table `a[i]` we use a simple algorithm. We fold from left to right the bounds. If two bounds are **always** equal, we didn't take the value between them, otherwise if the interval $[\![i; i + \text{len}]\!]$ overlapp with the interval created by the two bounds, we keep the argument. At the end, we doing the union of all segment we keep.

So we have :

$$\mathbb{E}^\#[\![a[i]]\!] = \bigcup_{j \in [\![0;2]\!]}^{\#} \left(\mathbb{E}^\#[\![s_{i+1}]\!] \circ \mathbb{C}^\#[\![b_j \leq i \leq b_{j+1}]\!]\right)$$

7.3.4. Example

In Listing 2, with your abstraction, we have the following state before the loop:

$$a = \{0\}\ \text{NOT\_INIT}\{b_1 = 0\}\ \text{NOT\_INIT}\{b_2 = \text{len}(a)\}\ \text{NOT\_INIT}\{\text{len}(a)\}$$

At the end of the first iteration, we have

$$a = \{0\}\ \text{INIT}\{b_1 = 1 = 2i + 1\}\ \text{NOT\_INIT}\{b_2 = \text{len}(a)\}\ \text{NOT\_INIT}\{\text{len}(a)\}\ \text{, with } i = 0$$

The widening operator is applied, resulting in $i = [1; 255]$. Then, when we execute `a[2*i] = 5`, we find that $2i = [2; 510]$. Thus, we have

$$a = \{0\}\ \text{INIT}\{b_1 = 1\}\ \text{MAYBE\_INIT}\{b_2 = \text{len}(a)\}\ \text{NOT\_INIT}\{\text{len}(a)\}$$

At the end of the loop, we can only conclude that index 0 is initialized, and possibly some indices between 1 and 255.

Note : Here we only discuss the initialization of the array. However, with the help of Mopsa and the separation of domains, we also have a range of possible values for each segments.

The only detail is that in Jasmin, we can cast an array to an array of the same size in bytes, but with a different type interpretation for values (see a table of `u64` to a table of `u32`, for example). In this case, for the moment, we send the $\top$ value in numerical domains to ensure soundness and avoid dealing with integer representation.

7.3.5. Concretization

We define the concretization function as follows: $\gamma_{\text{Arr}}(a[i]) = \gamma\big(\mathbb{E}^{\#}[\![a[i]]\!]\big)$. With the $\cup$ in the setter and getter, we lose some precision, but we are sure that all possible values are represented. And in practice, this covers the majority of initialization cases.

8. Contract and function call

Like in many programming languages, it is possible in Jasmin to make function calls. However, a function call does not have any side effects; that is, a Jasmin program does not produce any side effects. A function call takes a predefined number of arguments and returns a tuple of arguments that is immediately unpacked when we assign the result of the function call. This is written as follows in Jasmin: `v1, v2, v3 = f(a1,a2, a3,a4);` In this example, we take 4 arguments that return 3 values, but this number is not fixed.

We refer to a contract as a list of preconditions or postconditions that a function must satisfy. Mopsa already has a contract language for C (see [6]), which is similar to Frama-C's ACSL, but these are used only if the function source is not available. However, in the Jasmin team, we want to be able to check each function independently. Therefore, we slightly modified the behavior of contracts by adjusting our domain on your side.

The new behavior is as follows: When we have a function call, we check if there is a contract associated with this function. If so, we verify that the preconditions are satisfied and apply the postconditions to the variables on the left side of the assignment. If we do not find any preconditions, we only check that the scalar arguments are well-initialized, and we assign the return value to $\top$ and initialize it for the case of scalar variables.

Jasmin has an experimental feature for contracts that allows them to be sent to Easycrypt or Cryptoline. We reuse the syntax tree to capture expressions of the following form, which we can translate to Mopsa stubs.

The supported contracts in Jasmin for this experimentation are:

- Conjunction of propositions
- `init_array(v : var, offset: int, len: int)` a proposition that affirms that the array v is well-initialized between the indexes offset and len
- Postconditions and preconditions

Remark : The contract language from the user side does not have a proposition to indicate that a scalar variable is initialized, as this is a precondition and postcondition that must be checked for every function

9. Memory

Warning: This part is not yet implemented in the safety checker.

The memory model in Jasmin is quite simple; it can be viewed as a large array.

In the previous safety checker, it was able to infer which regions of an array had to be initialized. However, this mechanism does not seem useful, and the constraints provided by the programmer are preferred to ensure that the programmer knows what they are doing. For this reason, we prefer that the programmer provides a contract through the Jasmin annotation system.

If we assume that each pointer provided by the programmer points to a distinct region (with no overlap between the regions defined by each pointer), we can reuse the 3-segment implementation that we previously used for arrays Section 7.3.

The planned contracts for the moment consist of three predicates:

- `init_memory(v: var, offset: int, len: int)` defines that the region $[v + \text{offset}; v + \text{offset} + \text{len}]$ is readable, indicating that it is an initialized region.
- `write_memory(v: var, offset: int, len: int)` defines that the region $[v + \text{offset}; v + \text{offset} + \text{len}]$ is writable.
- `assign_memory(v: var, offset: int, len: int)` defines that the region $[v + \text{offset}; v + \text{offset} + \text{len}]$ is initialized at the output of the function.

To check for out-of-bounds access in memory, i.e., access to locations where we cannot write, we can easily verify that we are within the bounds declared as readable. In Jasmin, the memory is allocated before the call, and generally, there are not many arguments to function calls, so the number of segments will be finite and not expensive. To check if we can write to a location, the same check as before can be performed.

The complicated part is checking if a region that is not initialized at the function call becomes well-initialized at some point during the execution of the call or at the end, to verify the `assign_memory` predicate. If we assume the strong assumption that each pointer points to a different region of memory without overlap, we can consider each pointer as an array of length $\max_{\{((\text{offset}, \text{len}) \in P_v)\}}(\text{offset} + \text{len})$ (with P_v being the set of pairs of offset and length that appear in a predicate related to the variable v). We can then set as initialized the regions that we know are initialized if they are on one side of the array that abstract v.

This method, which unfortunately has not been tested in practice, seems to handle the majority of cases. The only real problem arises from the inability to handle pointer aliasing.

10. Performance and Implementation

```
Analysis terminated successfully
Analysis time: 0.026s

✗ Check #5:
jasa/tests/test_array_access_error.jazz:18.2-28: error: Jasmin Array out of bounds

 18:  s4 = s.[u32 4*8*n+16+1:n];
      ^^^^^^^^^^^^^^^^^^^^^^^^^^

 maybe a out of bounds with s at index ((((4 * 8) * 4) + 16) + 1)

Checks summary: 5 total,      4 safe, ✗ 1 error (selectivity: 80.00%)
 Jasmin Array out of bounds: 5 total,      4 safe, ✗ 1 error
```

Listing 3: Example of output of the safety checker when there is an error or a warning

To give an idea of the performance of the new safety checker, we tested it on the `ntt` function Section 14.2.

With the old safety checker, the analysis of the function took `30 seconds`; this is now possible in less than `3 seconds`. However, in both cases with the widening, we are only able to prove that the array is well-initialized (because it is well-initialized beforehand). Due to the upper bound approximation of scalars in the loop for the widening, we are unable to prove that there is no out-of-bounds access.

If we unroll the loop, we can prove that the array is well-initialized, that there is no out-of-bounds access, and that the loop terminates (as the analysis terminates with loop unrolling). This takes `45 seconds` with the new safety checker. With the older safety checker, after `3 hours`, the analysis was not finished.

In another file with 6 different functions[1], the analysis takes around `0.430 seconds` for the new checker and around `6.37 seconds` for the previous checker. (See Section 14.3 for details about the tests.)

The final implementation for the arrays feature consists of around 3,000 lines of code. The previous safety checker, which performed more checks, had 7,000[2] lines of code. However, around 1,000 lines of code in the new safety checker are dedicated to extending the MOPSA AST and translating the Jasmin AST to MOPSA.

11. Conclusion and future work

This new safety checker, written with MOPSA, checks that scalar variables are well-initialized, that array accesses are in bounds, and successfully verifies that arrays are well-initialized when they are initialized from one border to the middle. Moreover, the support for contracts allows for independent function checks and property verification, such as the initialization of arrays. Furthermore, with the modularity of MOPSA, the code is simple to extend and maintain. This implementation is also faster than the previous one, and the way MOPSA is built provides ideas for adding more checks inside the safety checker.

However, implementing support for basic memory will be necessary to accommodate a larger proportion of Jasmin programs. Without a doubt, it is possible to implement this with MOPSA without significant loss in speed.

It seems that with MOPSA, it is possible to handle more checks on values, such as verifying if an array is initialized with only 0s.

Detecting loop termination will also be interesting to check, but due to the over-approximation approach of MOPSA, this may not be easy to implement. Currently, termination is only proven when loops are fully unrolled, and the analysis terminates.

Other tasks need to be completed to finish the prototype and move towards a more “production-ready” product.

In Jasmin, it is possible to call a CPU instruction and get the result and flags, which depend on the target CPU. For now, the output is the top value and initialized for scalar values, but an approach similar to function calls will be possible, with specific cases handled. Similar functionality can also be implemented for system calls, which are not yet handled.

For a more advanced future, it is also planned to communicate to EasyCrypt which properties have already been proven by the safety checker, to establish a workflow where developers run the safety checker and subsequently prove parts in EasyCrypt that have not yet been verified.

[1] https://github.com/LeoJuguet/jasmin/blob/cryptoline-mopsa/compiler/jasa/tests/test_poly.jazz

[2] calculate with cloc, the code can be found here https://github.com/LeoJuguet/jasmin/blob/cryptoline-mopsa/compiler/jasa/

12. Acknowledgements

Thank you to the MPI-SP and the Gilles's Group for hosting me during this interniship. Thank you to the Formosa group for their hospitality. Thanks to Manuel Barbosa for his supervision and to Benjamin Grégoire, Vincent Laporte and Lionel Blatter for following the project and answering questions about Jasmin. Thanks to Raphaël Monat from the MOPSA team for taking the time to answer my questions about abstract interpretation and MOPSA and for monitoring the project.

13. Meta-Information

- 1.5 months were spent understanding the safety conditions of Jasmin and starting to grasp MOPSA, along with a first implementation to detect division by zero.
- 0.5 months were dedicated to checking for array initialization.
- 0.5 months were spent implementing the initialization of scalar values.
- 1 week was allocated for contracts and function calls.
- 1.5 months were used to understand the segmentation functor and attempt its implementation.
- 2 weeks to implement the final abstraction of arrays.

Some of the time was also spent fixing bugs and attending talks at the institute. Additionally, I had the opportunity to participate in two seminars with the Formosa team, where I met other team members in person and discussed safety and the future of Jasmin.

Bibliography

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017. Association for Computing Machinery, Dallas, Texas, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

[2] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, 1977. Association for Computing Machinery, Los Angeles, California, 238–252. https://doi.org/10.1145/512950.512973

[3] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, 2011. Association for Computing Machinery, Austin, Texas, USA, 105–118. https://doi.org/10.1145/1926385.1926399

[4] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. *SIGPLAN Not.* 46, 1 (January 2011), 105–118. https://doi.org/10.1145/1925844.1926399

[5] R. Monat. 2021. Static type and value analysis by abstract interpretation of Python programs with native C libraries.

[6] Abdelraouf Ouadjaout and Antoine Miné. 2020. A Library Modeling Language for the Static Analysis of C Programs. In *Static Analysis*, 2020. Springer International Publishing, Cham, 223–247.

14. Appendix

14.1. Test unrolling loop

```
fn f() -> reg u64
{
  reg u64 a;
  inline int i;

  for i = 0 to 256 {
    a = i;
  }

  return a;
}
```

For this pretty simple function that is safe and have only two simple scalar variabke, if we unroll the loop, the safety checker take `0.255s` but if we don't unroll the loop the same analysis take `0.007s`. So for a function simple like that the difference is important.

14.2. Ntt function

```
fn _poly_ntt(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
requires {init_array(0,0,KYBER_N)} // pre condition, rp is well initialized
ensures {assigns_array(0,0,KYBER_N)} // post condition, rp is well initialized
{
  reg u64 len;
  reg u64 start;
  reg u64 j;
  reg u64 cmp;
  reg u64 offset;

  reg u16 zeta;
  reg u16 t;
  reg u16 s;
  reg u16 m;

  reg ptr u16[128] zetasp;
  reg u64 zetasctr;

  zetasp = jzetas; // jzetas is a global array of type reg ptr u16[128]
  zetasctr = 0;
  len = 128;
  while (len >= 2)
  {
    start = 0;
    while (start < 256)
    {
      zetasctr += 1;
      zeta = zetasp[(int)zetasctr];
      j = start;
      cmp = start + len;
      while (j < cmp)
      {
        offset = j + len;
        t = rp[(int)offset];
        t = __fqmul(t, zeta);
        s = rp[(int)j];
        m = s;
        m -= t;
        rp[(int)offset] = m;
        t += s;
        rp[(int)j] = t;
        j += 1;
      }
      start = j + len;
    }
    len >>= 1;
  }

  rp = __poly_reduce(rp);

  return rp;
}
```

Listing 4: `ntt` function used for the performance test in Section 10, the difficulty to prove this function come from nested loop and the difficulty to the numerical domain to determine the relation betwen `len`, `start` and `j`, with the shift of `len`, so out of bounds access is detected without loop unrolling.

14.3. Details of performances test

The test was executed on a machine with:

- CPU: `AMD Ryzen 7 7840HS w/ Radeon 780M Graphics (16) @ 5.137GHz`
- GPU: `Radeon 780M`
- Memory: `16GB DDR5 5600MHz`
- OS: `Linux`

To test the performance of the new safety checker, we ran the following command in the `compiler` folder of the project:

```
time ./jasa.exe -config jasa/share/config_default.jazz file_test.jazz
```

The additional argument `-loop-full-unrolling=true` was added to perform the test with full unrolling.

To test the old safety checker, we ran the following command:

```
time jasminc -checksafety file_test.jazz
```

Before running this test, we verified that all functions were marked for export.

To test unrolling with the old safety checker, we generated a config file with `-safetymakeconfigdoc` and modified the `k_unroll` argument to `10000`, which simulates a large amount of loop unrolling.

15. Mopsa implementation

```
(* Section 3.3.2.2 *)
type ('a, 't) man = {
  get : 'a -> 't;
  set : 't -> 'a -> 'a;

  lattice : 'a lattice;

  exec : stmt -> 'a flow -> 'a post;
  eval : expr -> 'a flow -> 'a eval;

  ask : ('a,'r) query -> 'a flow -> 'r;
  print_expr : 'a flow -> (printer -> expr -> unit);
  get_effects : teffect -> teffect;
  set_effects : teffect -> teffect -> teffect;
}

 (* Section 3.3.2.4 *)
type 'a post = ('a, unit) cases
type 'a eval = ('a, expr) cases

module type DOMAIN =
sig
 (* Section 3.3.2.1 *)
  type t
  val id : t id
  val name : string
  val bottom: t
  val top: t
  val is_bottom: t -> bool
  val subset: t -> t -> bool
  val join: t -> t -> t
  val meet: t -> t -> t
  val widen: 'a ctx -> t -> t -> t

  (* Section 3.3.2.5 *)
  val init : program -> ('a, t) man -> 'a flow -> 'a flow
  val exec : stmt -> ('a, t) man -> 'a flow -> 'a post option
  val eval : expr -> ('a, t) man -> 'a flow -> 'a eval option

  (* Section 3.3.2.6 *)
  val merge: t -> t * effect -> t * effect -> t
  val ask : ('a,'r) query -> ('a, t) man -> 'a flow -> 'r option
  val print_state : printer -> t -> unit
  val print_expr : ('a,t) man -> 'a flow -> printer -> expr -> unit
end
```

Listing 5: Domain signature of MOPSA, section comment refer to sections of [5]