Jasa : The new Jasmin Safety Analyser write with MOPSA

Juguet Léo

March - Jully 2024

Abstract

We have written 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. This new safety checker is undoubtedly faster than the previous one. This new safety checker, created with MOPSA, also opens up the possibility to check more properties than the 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. The 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 static analyzer, but the previous one was too slow. The goal of this internship was to create a new, more efficient static analyzer that would be able to check that the conditions on a Jasmin code are well-present.

For this, I used MOPSA, a static analyzer library that aims to be modular, in order to easily add a frontend for the Jasmin language.

2. Context

Writing a secure program is hard, there are many considerations to take care of, such as ensuring that a variable is well-defined and that there are no out-of-bounds accesses.

In Jasmin, a safety analyzer already exists, but there is a main problems: The current implementation is to slow for big programs.

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 a function 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 2
4. Overview of Abstract Interpretation 3
   4.1. Widening 4
5. Overview of MOPSA 4

5.1. Semantic of statements 5
5.2. Default Domain 5
6. Initialisation of scalar 5
7. Initialisation of arrays 6
7.1. The difficulty 6
7.2. Aproach 6
7.3. 3 Segments 7
7.3.1. Representation 7
7.3.2. Setter 7
7.3.3. Access 8
7.3.4. Example 8
7.3.5. Soundness 9
8. Memory 9
9. Contract and function call 9
10. Performance and Implementation 9
11. Future 10
12. Conclusion 11
13. Acknowledgements 11
Bibliography 11

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. However, the current safety checker is too slow if we want to analyze a significant program.

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
- pointers that point to a memory space

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

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

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 operatorsa.

4. Overview of Abstract Interpretation

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.

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 $c \subseteq \gamma(a)$

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

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

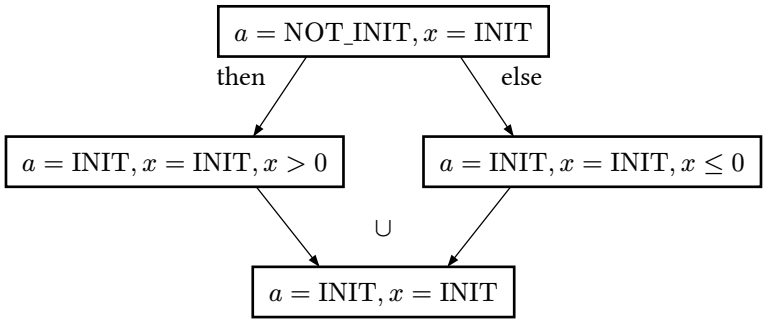

Figure 1: Hass diagram of Init poset

Definition 4: 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.

Definition 5: 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 $\forall a \in A, f(\gamma(a)) \subseteq \gamma\big(f^{\#}(a)\big)$

4.1. Widening

The widening operator was initially introduced in a Cousot's Cousot's paper [2]. The widening is the new feature that significantly improves the performance of the safety checker.

The principle of the widening is to find a stable state that is sound without unrolling loops.

Definition 6: widening
A binary operator $\triangle$ is defined by
- $\triangle : A \times A \to A$
- $\forall (C, C') \in A^2, C \leq C \triangle C', C' \leq C \triangle C'$
- $\forall (y_i)_{i \in \mathbb{N}} \in A$, the sequence $(x_i)_{i \in \mathbb{N}}$ computed as $x_0 = y_0, x_{i+1} = x_i \triangle y_{i+1}, \exists k \geq 0 : x_{k+1} = x_k$

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.

```
while i < 5 {
  i += 1;
}
```

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.

(**TODO: Insert an image to represent MOPSA**)

In MOPSA, to register a new abstraction, the developer only needs to register a Domain.

MOPSA performs an induction on the syntax.

MOPSA offers different domains that are already implemented for a universal language, and these domains can be easily used by another target language.

To have a correct abstract interpretation with MOPSA, we only need to provide a domain of the appropriate form.

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 $\triangle$
- a concretisation operator $\gamma : D^{\#} \to P(\mathbb{Z})$

5.1. Semantic of statements

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

$$\mathbb{E}[\![v]\!]\sigma = \{\sigma(v)\}$$
$$\mathbb{E}[\![z \in \mathbb{Z}]\!]\sigma = \{z\}$$

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.

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

$$\mathbb{S}[\![[s_1; ...; s_n]]\!] = \mathbb{S}[\![s_n]\!] \bigcirc ... \bigcirc \mathbb{S}[\![s_1]\!]$$
$$\mathbb{S}[\![\text{if } c\{s_t\} \text{ else } \{s_f\}]\!] = \mathbb{S}[\![s_t]\!] \bigcirc \mathbb{C}[\![c]\!] \cup \mathbb{S}[\![s_f]\!] \bigcirc \mathbb{C}[\![\neg c]\!]$$
$$\mathbb{S}[\![x = e]\!]\Sigma = \{\sigma[x \leftarrow v] \mid \sigma \in \Sigma, v \in \mathbb{E}[\![e]\!]\sigma\}$$

These statments are classic in abstract interpration so Mopsa, already have domain that handle theses statments.

5.2. Default Domain

Mopsa offer different default domain :

- numerical domains : in particular interval domains and relational domains.
- iterators domains in universal language to manipulate principal statements.

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 following abstract value domain Figure 2.

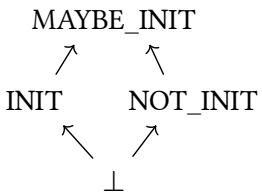

Figure 2: Poset of Init domain

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

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

$$\mathbb{E}[\![a \star b]\!] = \mathbb{E}[\![a]\!] \cup \mathbb{E}[\![b]\!] \text{ whith} \star \in \{+, -, *, /, \%\}$$
$$\mathbb{E}[\![\bullet\, a]\!] = \mathbb{E}[\![a]\!] \text{ whith} \bullet \in \{+, -\}$$
$$\mathbb{E}[\![c]\!] = \text{INIT with } c \text{ a constant}$$

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

This is a value abstract domain. So we can implement it in Mopsa and have a correct abstract interpretation of the initialisation of scalar variables.

7. Initialisation of arrays

We name a cell initialised if before the cells was assign to a well initialized expression.

7.1. The difficulty

The principal difficulty to prove that an array is well initialized, is to be sure that the overapproximation of the index use to set a cell in a loop will not provided an approximation were at the end all the array is initialized, when it's false like in the Listing 1 were the over approximation will give `a[x] = 5` with $x \in [2; 512]$, but only even indexes are initialized.

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

Listing 1: partial initialisation of an array

7.2. Aproach

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.

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.

Cousot's Cousot's approach: 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, or to be able to do a symbolic equalities of a variable in two different context, but this ask to modify Mopsa.

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 to the center, this initialization is a form of array partitionning but with a fixed number of partition at 3. In pratices, this is the case in a majority of cases that jasmin deal.

7.3. 3 Segments

7.3.1. Representation

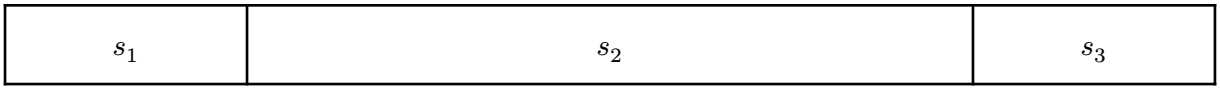

Figure 3: Representation of an array

We represent each arrays with 3 segments, so only 7 variables are needed to represent an array. 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 :

$$\begin{aligned} b_0 &= 0, \\ b_1 &= 0, \\ b_2 &= b_3, \\ b_3 &= \text{len}(a) \\ e_1 = e_2 = e_3 &= \top \text{ and not init} \end{aligned}$$

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 3 segments, has two "movable" bounds b_1, b_2, in lot of cases, the center is not initialized. So the extremities contains more precise information. We want avoid to lose it, so we extends the extremities segments to the middle only when we are sure that the bounds is always included in the interval set.

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 cases, if different cases are possible, if there exist in the environment a concretisation such that the cases is valid, we do all cases separately and we join the different array abstraction obtains.

$$\text{if } [i; i+\text{len}] \subseteq \left[b_j; b_{j+1}\right] \text{ then } s_{j+1} = s_{j+1} \cup e (\text{with } j \in [\![0;2]\!])$$

$$\begin{cases} \text{if } b_1 = i \text{ then } \begin{cases} \text{if } i+\text{ len} \leq b_2 \text{ then } b_1 \leftarrow i+\text{ len} \wedge s_1 \leftarrow s_1 \cup e \\ \text{if } i+\text{ len} > b_2 \text{ then } b_1 \leftarrow i+\text{ len} \wedge b_2 \leftarrow i+\text{ len} \wedge s_1 \leftarrow s_1 \cup e \\ \text{if } i+\text{ len} = b_2 \text{ then } b_1 \leftarrow i \wedge b_2 \leftarrow i+\text{ len} \wedge s_2 \leftarrow s_1 \cup e \end{cases} \\ \text{if } b_1 < i \text{ then } \begin{cases} \text{if } i+\text{ len} < b_2 \text{ then } s_2 \leftarrow s_2 \cup e \\ \text{if } b_2 = i+\text{ len then } b_2 = i \wedge s_3 \leftarrow s_3 \cup e \\ \text{if } b_2 < i+\text{ len then } \begin{cases} \text{if } b_2 < i \text{ then } s_3 \leftarrow s_3 \cup e \\ \text{if } b_2 \geq i \text{ then } b_2 = i \wedge s_3 \leftarrow s_3 \cup e \end{cases} \end{cases} \\ \text{if } i < b_1 \text{ then } \begin{cases} \text{if } i+\text{ len} < b_1 \text{ then } s_1 \leftarrow s_1 \cup e \\ \text{else } b_1 \leftarrow i+\text{ len} \wedge s_1 \leftarrow s_1 \cup e \end{cases} \end{cases}$$

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 in maths way we have : $\gamma(a[i]) = \cup_{j \in [\![0;2]\!] \wedge \gamma(i) \cap \gamma([b_j; b_{j+1}]) \neq \emptyset} \gamma(s_j)$

7.3.4. Example

Example

```
fn f() -> reg u64
{
  reg u64[20] a;
  inline int i;
  reg u64 ret;
  reg bool b;

  b = true;
  i = 0;

  while (b) {
    if i > 10 {
      b = false;
    }
    a[i] = 0;

    i += 1;
  }

  ret = a[15];
  return ret;
}
```

On the example on the right, after that the first loop is executed, we have $a = \{0\}$ INIT $\{b_1 = 1\}$ NOT_INIT $\{b_2 = 21\}$ NOT_INIT $\{b_3 = 21\}$ and $i = 1, b = \text{true}$, before redo a loop iteration, the widening operator is apply, and the numeric domain cannot infer that $i \leq 10$, will we have the upper approximation $i = [1; +\infty]$, and we also have that $i = b_1$.

$a = \{0\}$ INIT $\{b_1 = [1, +\infty]\}$ NOT_INIT $\{b_2 = 21\}$ NOT_INIT $\{b_3 = 21\}$ and $i = 1, b = \text{true}$,

So when we access at the index 15 of the array a we do the join of s_1 and s_2 so we obtain MAYBE_INIT.

Note : Here we only talk about initialization of the array, but in fact, with the help of Mopsa, and the separation of domains, we also have a range of possible values for each interval. The only details is that in Jasmin, we can cast an array to array of the same size in byte, 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 $\top$ value in numerical domains, to be sure to be sound and avoid to deal with intenger representation.

7.3.5. Soundness

8. Memory

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

The memory model in Jasmin is pretty 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. But this mechanism does not seem useful, and the constraints given by the programmer are preferred, to ensure that the programmer knows what they are doing. For this, we prefer that the programmer provides a contract Section 9 through the Jasmin annotation system.

If we assume that each pointer provided by the programmer points to a distinct region (there is 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 3 predicates:

- `init_memory(v: var, offset: int, len: int)` defines that the region $[v + \text{offset}; v + \text{offset} + \text{len}]$ is readable, so 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.

To check for out-of-bounds access in memory, i.e., access to places where we cannot write, we can easily check 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 place, the same check as before can be done. The complicated part is to check if a region that is not initialized at the function call but is well-initialized at some point during the execution of the call or at the end. So, to check the `assign_memory` predicate. If we assume the strong assumption that each pointer points to a different region of memory without overlapp, then 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 the set of pair offset, length that appear in a predicate relatif to v) and we can set as initialized the regions that we know are initialized, if they are on one side of the array.

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

9. Contract and function call

Like in lot of programming language, it's possible in Jasmin to do function call, but a function call didn't have any side effect, like jasmin program didn't produce any side effect. A function call take differents a predefined number of argument and return a tuple of argument that is imediately split when we do an assign of a function call this is write like this `v1, v2, v3 = f(e1,e2,e3);` in jasmin.

When we have a function call, we check if we have a contract associated to this function, if yes we check that the requires are satifies, and we apply the ensures to the variable v_i. Otherwise if we didn't found any requires we only check that the scalar in argument are well initialized, and we assign the return value to $\top$ and initialized for the case of scalar variable.

10. Performance and Implementation

```
Analysis terminated successfully
Analysis time: 0.005s

    Check #4:
jasa/tests/test_multiples_files/test_files2.jazz:12.1-14: warning: Jasmin Scalar Initialization

 12:    c = add(a,b);
        ^^^^^^^^^^^^

 'T:T_J_U64' is not initialized


✗ Check #8:
{jasa/tests/test_multiples_files/test_files2.jazz}::return expression of f: error: Jasmin Scalar Initialization


 'c' is not initialized

Checks summary: 8 total,      6 safe, ✗ 1 error,      1 warning
 Jasmin Scalar Initialization: 8 total,      6 safe, ✗ 1 error,      1 warning
```

To give an idea of the performance of the new safety checker, we tried 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 before). Due to the upper bound approximation of scalars in the loop for the widening, we are not able to prove that there is no out-of-bounds access. If we unroll the loop, we are able to prove that the array is well-initialized and there is no out-of-bounds access, 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 `0.430s` for the new checker to be analyzed, and around `6.37s` for the previous checker. (see Section 14.3 for details about the tests.)

The final implementation for the arrays feature counts around 3,000 lines of code. The previous safety checker that performed more checks has 7,000 lines of code. However, around 1,000 lines of code in the new safety checker are for extending the MOPSA AST and translating the Jasmin AST to MOPSA.

11. Future

This new safety checker, written with MOPSA, is faster than the previous one, and the way MOPSA is built provides some ideas to add more checks inside the safety checker. However, implementing the support of basic memory will be necessary to support a larger proportion of Jasmin programs. But 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.

Detection of 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 proved 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”.

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

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 things can also be done for system calls, which are not yet handled.

12. Conclusion

The new safety checker written with MOPSA is faster than the previous one and offers the possibility of carrying out more checks in the future. Futhermore, Mopsa's modularity means that the code is more readable, faster and easier to add options to. For the moment, the checker correctly checks variable initialisation, array initialisation in most cases and out-of-bounds accesses. In addition, it's modular and works using a contract system.

13. Acknowledgements

Thank you to the MPI-SP and the Groupe de Gilles for hosting me during this course. 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 Raphael Monat from the MOPSA team for taking the time to answer my questions about abstract interpretation and MOPSA and for monitoring the project.

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.

14. Annexes

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)}
ensures {assigns_array(0,0,KYBER_N)}
{
  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;
}
```

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.