Compiling Hermes to RSSA

Bernard Legay Halfeld Ferrari Alves

Department of Computer Science, Copenhagen University, Copenhagen, Denmark

Abstract

Reversible programming languages have been a focus of research for more than a decade, mostly due to the work of Glück, Yokoyama, Mogensen, and many others. In this paper, we report about our recent activities to compile code written in the reversible language Hermes to reversible static-single-assignment form RSSA. We will also discuss how we wrote an interpreter for an extended version of RSSA using a type system. Our compiler allows the execution of simple Hermes programs and provides the basis for further optimizations.

Keywords: Reversible computing · Reversible programming languages · Hermes · Reversible static-single-assignment . Encryption

1. Introduction

Reversible computing has been a major research area over the last decade. Indeed, through the ability to use the same code to execute programs forward and backward, reversible computing has been seen as beneficial for reducing energy consumption, by implementing only one function in order to execute both it and its inverted variant.

While many reversible languages have been developed throughout the years, the language Hermes [9] stands as an important advancement of the field. Developed mainly by Mogensen, this language is intended to be used for encryption and, as such, addresses concerns found in other reversible languages such as Janus, separating secret and public data and restricting operations on secret data to those whose time do not depend on the actual values processed.

However, while much advancement has been done on Hermes, the question of optimization still prevails as one would want to improve the execution time of programs or their memory consumption. It is well known that optimization can most effectively be performed on some intermediate representation of the source rather than the source code itself. This includes static-single-assignment.

Developed by Mogensen, RSSA [6] is a special variant of static-single-assignment which can be executed forwards and backwards. This language has also been used in order to optimize other reversible languages such as Janus [4], through the use of a compiler from Janus to RSSA. [3]

In this paper, we will present our work which involves the implementation of a compiler capable of compiling Hermes into RSSA programs. In order for such a compiler to be created, we have also implemented an interpreter for an extended version of RSSA, which takes into consideration some specificities from Hermes, allowing for a smoother transition from one language to the other.

In this context, the main contributions of our work are:

- Implementation of an extended version of RSSA, considering types and size of variables and creating new statements to change types.
- Implementation of an interpreter for the extended version of RSSA.
- Creation of a compiler of Hermes for this extended version of RSSA. The main challenge here has been the definition of the correspondence between the Hermes procedures and the collection of blocks in RSSA that would implement the same function.

The rest of this work is organised as follows. In Section 2, we will begin with a brief explanation of both Hermes and RSSA in order to provide a basis for the description of our compiler. Then in Section 3, we will describe our interpreter and compiler, as well as the extended version of RSSA we came up with. In Section 4, we will also present the steps to translate Hermes into our extended RSSA, in order for our compiler to be operational. Finally in Section 5, we will test our compiler on various example codes and compare the results to the Hermes interpreter.

2. Preliminaries - Hermes and RSSA

2.1 Hermes

Hermes was first introduced by Mogensen [9] with the intent to be used for encryption. As such it features some unique characteristics compared to other reversible languages.

Most prominently is the typing system of Hermes. While every variable is an unsigned integer, they all possess different bit sizes, which is represented through a different type. A variable of type *u8* will have 8 bits, *u16* 16 bits, *u32* 32 bits and finally *u64* will have 64 bits. Additionally, each variable is specified if they are secret or public, represented by either *secret* or *public* before their types. So *public u64* represent a public variable of 64 bits. Constants are implicitly a public 64-bit number. Operations with only public values will have public results, and similarly for secret values. When public and secret values are mixed, the result will be secret.

Both scalar and array variables are used in Hermes. For arrays in particular, the array variable must have an array type and the index expression must be public. This ensure that no secret information is leaked during timing of memory accesses. Conversely, unsafe arrays can use both public and secret indexes, however their elements must be secret.

A Hermes program is composed of procedures. Each procedure contains statements. While a procedure can contain a single statement, multiple statements must be grouped with brackets, indicating the start and end. The program is also reversible, and can be inverted by inverting every statement and their order.

In Figure 1 is an example of a Hermes Program. The syntax used in this example is an extended version of the one described in Appendix A. The <−> symbol signifies a swap of value between two variables. <<= and >>= are left and right rotates respectively, meaning the value of the variable on the left has its binary value shift by the amount of the variable on the right, either left or right. i++ is the equivalent to i += 1. Finally, @ means that the statement on the left is reversed once the statement on the right has been executed. During parsing, an statement of the form $c1@c2$ is expanded into $c1c2\bar{c1}$, where $\bar{c1}$ is the inverse of $c1$. This operator is right associative. More examples can be found in Appendix E.

```
rc5(u32 ct[], u32 S[])
{
  u32 A, B;
  A <-> ct[0]; @ B <-> ct[1]; @
  {
    A += S[0]; B += S[1];
    for(i=2; size S) {
      A ^= B; A <<= B; A += S[i];
      B <-> A;
      i++;
    }
  }
}
```

Figure 1: Example of a Hermes Program

2.2 RSSA

Reversible static-single-assignment (RSSA) was firstly introduced by Mogensen [6] as a reversible variant of SSA (Static-Single-Assignment), which in turn is an intermediate language to facilitate data-flow analysis and was proposed by Alpern, Wegman, and Zadek in 1988 [1]. The representation of SSA is such that each variable has only one definition and each update to the variable creates a new variable instead. The syntax of RSSA can be found in Appendix B.

RSSA uses variables and constants defined as atoms. These atoms are all typed as integers. It also allows access to a memory location in the form $M[a]$, representing the location to where an atom a points. Atoms and Memory accesses can be used in conditions.

In RSSA, a program is defined as a set of basic blocks, each consisting of a sequence of assignments or a call. Each block is enclosed by an entry and exit point. These entry and exit points use labels that must be utilized at exactly one entry and exit point. These labels, and the associated parameters, replace the Φ-functions used in the original SSA form. Each RSSA program must possess one entry point *begin main* and a corresponding exit point *end main*. Entry and exit points may be either conditional or unconditional.

For conditional entry points, they are represented as such: $L1(x,...)L2 \leftarrow C$ where C is the condition and L1 and L2 the different labels. The result of the condition will determine through which label the jump entered, L1 if C is true and L2 if it is false. The exit jump has a similar manner: $C \rightarrow L1(x,...)L2$. The condition is evaluated and depending on the evaluation a jump to either L1 if true or L2 if false will be made.

Variables declared in entry points, whether conditional or unconditional, will be given the value of the parameter of the incoming label.

RSSA defines a set of reversible instructions, that can be used to compose reversible programs. Every instruction has a well-defined reverse. The program can be inverted by inverting every instruction in it and their order.

3. Our Compiler

This section briefly explains the structure and implementation of our compiler. In Section 3.1, we give some details on the changes made to RSSA to obtain its extended version. In Section 3.2, we give an overview of our RSSA interpreter. In Section 3.3, we briefly describe our compiler, indicating the modules used from the interpreters of both languages.

3.1 Extension of RSSA

Since Hermes is designed to be used for encryption, some of its elements cannot be directly translated into RSSA. Because of this, we have decided to extend the original form of RSSA in order to allow the Hermes specific forms to be used. The syntax for this extended version of RSSA can be found in Appendix C. Below, we will discuss some of the most significant changes.

3.1.1 Type system

The first aspect that needs to be added is the type system. For this, we decided to take a similar approach to Hermes, using types to determine the size and status of each variable. To simplify, we decided to represent the type in the form uZ where u can either be s or p, representing secret and public respectively, and Z the size, either 8, 16, 32 or 64. These types are placed before the introduction and elimination of the variable as well as in entry and exit jumps and function calls, in order for the program to be reversible. When using the variable in a condition or in an update, the specification of the type is not needed as we assume the program already knows their type.

The addition of types also require us to create type rules for RSSA. These rules are similar to the ones in Hermes, with some changes in order to accommodate the different structure. We also use an environment ρ to store each variable that is used and their types, allowing the program to verify the type of a variable if needed. When a variable needs to be destroyed, that variable is removed from the environment. We also use a Γ in which we include the labels of all jumps as well as a boolean indication to check if it is an entry or an exit jump. This Γ is used during calls and jumps in order to determine which block needs to be executed next. These different type rules are defined in Appendix D.

3.1.2 Memory

Another aspect of Hermes that must be added is the array access, since RSSA doesn't naturally have any arrays. To do this, we use the memory access in order to replicate a similar effect. Since arrays are also typed in Hermes, the same must also be done for the memory access, using the same model as the regular variables. We will also use two different memories, one for secret and one for public variables, in order to avoid leaks from secret to public arrays. The addresses of these memories are always public, in a similar way to Hermes.

3.1.3 Reveal/Hide

We also need to be able to change the types of variables, turning them from public to secret and inversely. This would allow us to temporarily use a secret value as an address, by changing its type and reverting it back once the operation is complete. In order to accomplish this, we have introduced two new statements in our extended RSSA version, reveal and hide. Reveal allows secret variables to be used as public variables, by redefining a new public variable and uncomputing the old secret one. Conversely, hide allows public variables to be used as secret, creating the opposite effect. Below is a simple example of how both reveal and hide are used.

```
p64 x1 := reveal s64 x0
s64 y1 := hide p64 y0
```

Figure 2: Exemple of the use of Reveal and Hide in our extended RSSA version

3.2 Interpreter for extended RSSA

This subsection discusses the interpreter made for the extended RSSA language above. The interpreter is composed of different modules which allow the validation of the RSSA code and its execution. The modules are implemented in Standard ML, using Moscow ML in order to compile them.

First the lexer handles the lexical analysis of the code, transforming the different strings of the code into appropriate tokens. Next, the parser executes the syntactic analysis, verifying that the grammar of RSSA is respected and uses the tokens to construct a abstract syntax tree. This syntax tree is defined in a file named Data.sml, which also holds the code in order to invert the program as well as a print function to see the generated program on a terminal. Next, the type checker verifies if the type rules we defined are respected. Finally, the interpreter executes the program, either normally or in reverse depending on the desired effect. The inputs of the main function are given by the user, as well as the initial secret and public memories, and the program returns the output of the main function, and the secret and public memories, showing any potential modifications they may have had.

For the normal execution of the program, the interpreter runs the acquired syntax tree as normal. However, when doing an reverse execution, the interpreter first reverses the commands. Firstly, the order is reversed, starting from the last command given in the program and finishing with the first. Secondly, the individual commands are reversed as well. Below is an example of a RSSA program, and its reversed variant.

```
begin main(p64 x0, p64 y0)
  p64 x1 := x0 + 5
  p64 x2 := x1 + (y0 * 2)
end main(p64 x2, p64 y0)
```

```
begin main(p64 x2, p64 y0)
  p64 x1 := x2 - (y0 * 2)
  p64 x0 := x1 - 5
end main(p64 x0, p64 y0)
```

Figure 3: Example of an RSSA program (Left) with its reversed variant (Right)

3.3 Compiler

The compiler is composed of various elements of both the Hermes and our RSSA interpreter. The Hermes interpreter used is the same as the one defined by Mogensen [9].

From the Hermes interpreter, we use its lexer, parser and type checker. Similarly to our interpreter, the Hermes lexer and parser translate a Hermes program into a syntax tree. The type checker is also used in order to make sure that the original code doesn't have any errors in regards to typing.

From our RSSA interpreter, we use the type checker and main interpreter. This is because the compilation will be done on the Hermes syntax tree, therefore the lexer and parser of RSSA are not needed for the compiler. The type checker verifies if the given program doesn't have any errors in regards to typing and the interpreter will execute the code in a similar manner to the original interpreter.

The compiler first translates the original Hermes code using the lexer and parser from the Hermes interpreter. It then translates the resulting Hermes syntax tree into a RSSA syntax tree using a compiler program. The details of this translation will be discussed below. The RSSA syntax tree is then used in the interpreter in order to execute the program. Both syntax trees are also passed through their respective type checkers.

4. Compiling Hermes to RSSA

When compiling Hermes to RSSA, the original code must be changed in order to be properly translated into our extended RSSA language. In this section, we will discuss how we translated some of Hermes' syntax into our extended version of RSSA.

4.1 Blocks

In Hermes, a function can contain either one or multiple statements. In the case of multiple statements, they are represented as a list inside of a block, outlined by opening and closing braces. For a single statement, it is only necessary to translate that statement. However, for a block of statements, all statements must be translated. The order in which they are presented in RSSA is the same as their original order in Hermes, as a way to maintain the same execution.

Blocks in Hermes can also be accompanied by declarations of new constants or variables. These are therefore translated into RSSA. For constants, they are given the specified value and the type p64. For variables, they are given the specified type and an initial value of 0. At the end of the block, all of these variables are uncomputed in order to preserve reversibility.

```
{
  u32 a;
  const CONST = 5;
  ...
}
```

```
s32 a0 := 0
p64 CONST0 := 5
...
p64 5 := CONST0
s32 0 := a0
```

Figure 4: Example of blocks and declarations in Hermes (Left) and RSSA (Right)

4.2 Variables

4.2.1 Variable Updates

The translation of simple updates is rather straightforward. To do so, we initialise a new variable and give it the value of the original variable with the corresponding operation of the update. The name of this new variable is the same as the original but with its numbered incremented, so if the original variable is $x0$, the new one will be $x1$. With operations that require two values or less, the update is translated into RSSA with a single update line.

```
x += y+2;                             p64 x1 := x0 + (y0 + 2)
```

Figure 5: Example of a simple update in Hermes (Left) and RSSA (Right)

For update operations that require more than two values, it is more complicated, since the syntax of RSSA doesn't allow complex expressions. In order to circumvent this, we create an other variable T, which takes the value of the embedded operation. This is done by doing a XOR operation between 0 and the embedded operation. T is then used in the update operation. Once the update is complete, T is eliminated in order to respect reversibility.

```
                                      p64 T0 := 0 ^ (2 * 3)
                                      p64 x1 := x0 + (1 + T0)
x += 1+2*3;                           p64 0 := T0 ^ (2 * 3)
```

Figure 6: Example of a complex update in Hermes (Left) and RSSA (Right)

4.2.2 Variable Swaps

For swaps, the translation will be different depending on the items being swapped.

If two variables are swapped, then a double assignment is necessary. The new variables, are given the value of the opposing old variables. For example, in Hermes if x and y are being swapped then, in RSSA, the resulting variables $x1$ and $y1$ will be given the values of $y0$ and $x0$ respectively.

```
x <-> y;                              p64 x1, p64 y1 := y0, x0
```

Figure 7: Example of a variable swap in Hermes (Left) and RSSA (Right)

4.3 Arrays

For properly translating arrays, some specific steps are required. Array variables are represented by two separate variables, one represents the size of the array, and the other the address of the start of the array. Both are given types p64 regardless of the type of the original array. When using the array at index i in the procedure, we first create a variable I0, which contains the addition of the address of the start of the array and i scaled to the element size. It is important to not destroy the original variable, as it could still be used later. We then use a memory access at index I0 in order to execute the desired operation. The memory that is chosen depends on the original type of the array, secret arrays will use the secret memory and public arrays the public memory. Once it is done, we simply uncompute I0.

Our memories are also byte-addressed, meaning that the index is scaled to each individual byte. Because of this, when using the index, a binary left shift must be done. The amount depends on the size of the original array. So if the array is of size 8, a left shift of 0 must be done on the index, if of size 16 a left shift of 1, size 32 with 2 and size 64 with 3. This shift is implemented into a variable T0 which is used during the operation of the array and uncomputed afterwards.

4.3.1 Array Updates

When updating an element in an array, I0 and T0 are used in order to replicate the update.

```
                                    p64 T0 := 0 + (x0 << 3)
                                    p64 I0 := 0 + (ctT0 + T0)
                                    p64[I0] += 1
                                    p64 0 := I0 - (ctT0 + T0)
ct[x] += 1;                         p64 0 := T0 - (x0 << 3)
```

Figure 8: Example of an array update in Hermes (Left) and RSSA (Right)

If an array is used during an operation, an additional line is required in RSSA. A new variable T1 is created and given the value of the element of the array that we require. This variable is created by using the memory and variable swap line, temporarily replacing the element in the memory by 0. We then use T1 in the required operation before uncommiting it. The element of the memory also regains its original value.

```
                                    p64 T0 := 0 + (y0 << 3)
                                    p64 I0 := 0 + (ctT0 + T0)
                                    T1 := p64[I0] := 0
                                    p64 x1 := x0 + T1
                                    0 := p64[I0] := T1
                                    p64 0 := I0 - (ctT0 + T0)
x += ct[y];                         p64 0 := T0 - (y0 << 3)
```

Figure 9: Example of an use of an array in Hermes (Left) and RSSA (Right)

4.3.2 Array-element Swaps

For swapping the values of two elements of arrays, weather it be two elements of the same array or two different arrays, we use the memory swap command of the grammar as well as the predefined T0 and I0 for both arrays.

For swapping the values of a variable with an element of an array, we use the swap command between a variable and a memory.

```
                                  p64 T0 := 0 + (x0 << 3)
                                  p64 I0 := 0 + (ST0 + T0)
                                  p64 T1 := 0 + (y0 << 3)
                                  p64 I1 := 0 + (ctT0 + T1)
                                  p64[I0] <-> p64[I1]
                                  p64 0 := I1 - (ctT0 + T1)
                                  p64 0 := T1 - (y0 << 3)
                                  p64 0 := I0 - (ST0 + T0)
S[x] <-> ct[y];                   p64 0 := T0 - (x0 << 3)
```

Figure 10: Example of an array element swap in Hermes (Left) and RSSA (Right)

```
                                  p64 T0 := 0 + (y0 << 3)
                                  p64 I0 := 0 + (ctT0 + T0)
                                  x1 := p64[I0] := x0
                                  p64 0 := I0 - (ctT0 + T0)
x <-> ct[y];                      p64 0 := T0 - (y0 << 3)
```

Figure 11: Example of swap between a variable and an array element in Hermes (Left) and RSSA (Right)

4.3.3 Size

When the size of an array is needed, we simply use the variable representing the size of the array as discussed earlier.

```
x += size ct;                     p64 x1 := x0 + ctS0
```

Figure 12: Example of use of array size in Hermes (Left) and RSSA (Right)

4.4 If-Else

Unlike Hermes, RSSA doesn't use any if-else statements. In order to replicate a similar effect, conditional jumps and joins are used instead.

When an if-else statement is made in Hermes, we create two new blocks, one for the then statement and one for the else statement. These blocks are linked together using a conditional jump. If the condition is true, the program will execute the then-block, if its false, it will execute the else-block. These blocks contain unconditional entry and exit jumps, as the condition has already been tested beforehand. The program then goes back into a third block which represents the rest of the function. This last block contains a conditional join, in order to allow both prior blocks to converge. The Hermes type checker ensures that any variable used in the condition cannot be modified while in the then or else statements, so the original condition will always have the same result at the end of the if-else statement.

The entry and exit points of two connecting blocks must have the same number of arguments. In order to simplify it, the entry and exit points pass all variables currently used in the function. Any temporary variable created in the then or else statements are automatically destroyed before the join. The names of the transferred variables are incremented in order to create new variables from them.

The entry and exit points are also required to contain labels, directing the program into the next block. Since these labels need to be unique throughout the entire program, we came up with a naming convention, assuring that each label has a distinct label. The labels for the conditional and unconditional jumps are of the form $label*incr*Lx$. $label$ is the name of the current function. In the case of indented if-else statements, we add either "if" or "else" to the label depending on where the indentation is. This allows the create multiple indented if-else statements. $incr$ is a number starting at 0 which is incremented once the statement is concluded. This allows for multiple if-else statements in the same function. Finally Lx represents the number of the label. x goes here from 1 to 4 and represents a link between two blocks. $L1$ and $L2$ are used in the first conditional jump and lead to the then and else blocks respectively. $L3$ and $L4$ are the join labels, reuniting both blocks at the end.

The conditions used in the conditional exit are the same as the ones in Hermes. If the Hermes condition already contains a comparison operator, the condition is used as is. Complex conditions require the use of additional variables which are computed prior to conditional jumps and are then uncomputed after the conditional join. If the Hermes condition a does not contain a comparison operator, the condition is turned into a != 0, since it is the closest equivalent to Hermes' interpretation. Again, complex conditions require additional variables which are computed before the conditional jump and then uncomputed after the conditional join if necessary.

```
                    y0 < 5 -> main0L1(p64 x0, p64 y0)main0L2
                    main0L1(p64 x1, p64 y1) <-
                      p64 x2 := x1 + 1
if(y<5){            -> main0L3(p64 x2, p64 y1)
        x++;        main0L2(p64 x3, p64 y2) <-
} else {              p64 x4 := x3 - 1
        x--;        -> main0L4(p64 x4, p64 y2)
}                   main0L3(p64 x5, p64 y3)main0L4 <- y3 < 5
```

Figure 13: Example of a if-else statement in Hermes (Left) and RSSA (Right)

Figure 14 represents the flowchart of the events happening when dealing with an if-else statement.

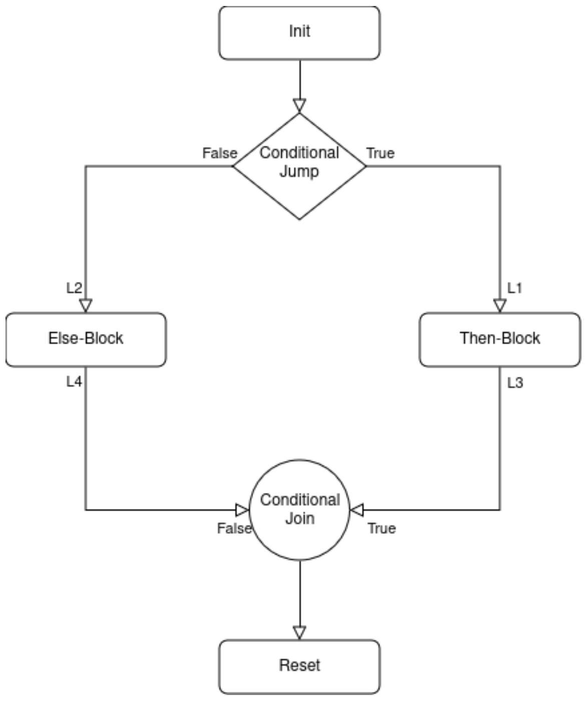

Figure 14: Flowchart representing the events of an if-else statement

4.5 For-loops

Similarly to if-else statements, for-loops are not used in RSSA. Its usage also requires the use of jumps.

This time, only one block is necessary to replicate the for-loop. Its entry and exit are both conditional. Both conditions used are in the form of $a == b$, in order to replicate the use of for-loops. Both have one label in common, which serves when the condition for the exit loop is false. This leads the program to execute the block again, thus replicating the effects of a for-loop. The other labels used are the links with the original block and the end block once the condition has been fulfilled.

The naming convention for the labels is similar to the if-else statements, allowing the use of multiple loops and indented loops.

A new variable i is always declared at the beginning of the loop. In RSSA, it will be committed at the beginning of the loop and uncommitted at its end. This variable is thus used for the conditions of the for-jumps. The entry jump uses the starting state of i and checks if i starts at the desired value. The exit jump uses the ending value of i, once it reaches this value, the program can pass to the next block. Similarly to Hermes, i is not automatically updated and thus, it must be specified during the loop in the original program. Additional variables are also committed if i takes the value of an operation or an element of an array. These same variables are also uncommitted at the end of the loop.

Figure 16 represents the flowchart of the events happening during the for-loop.

```
                           p64 i0 := 0 ^ 0
                           ->main0L1(p64 x0, p64 y0, p64 i0)
                           main0L1(p64 x1, p64 y1, p64 i1)main0L2
                               <-i1==0
                             p64 x2 := x1 + 1
                             p64 i2 := i1 + 1
for(i=0; y){               i2==y1->
        x++;                   main0L3(p64 x2, p64 y1, p64 i2)main0L2
        i++;               main0L3(p64 x3, p64 y2, p64 i3)<-
}                            p64 0 := i3 ^ y2
```

Figure 15: Example of a for-loop in Hermes (Left) and RSSA (Right)

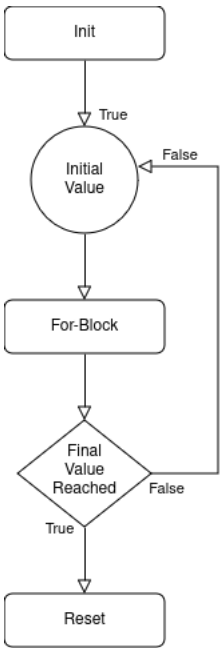

Figure 16: Flowchart representing the events of an for loop

4.6 Unsafe Arrays

Similarly to arrays, specific steps are required in order to properly translate unsafe arrays in RSSA. The use of an unsafe array is similar to how regular arrays are used.

The difference is that the variable IO, used as the index of the array must have a secret type, instead of a public one. Additional temporary variables used are also given secret types. We then introduce a second variable $I1$ which has a public type. $I1$ is given the value of $I0$ using the reveal statement described earlier, and $I1$ is then used as the index to the appropriate memory. The choice of memory, as well as the handling of the statement thereafter, is the same as with regular arrays. Once the operations have been made, we introduce a third variable $I2$, a secret variable which is given the value of $I1$ using the hide statement. $I2$ is then uncomputed. The last two steps are done in order to preserve reversibility.

```
                                        s64 T0 := 0 + (z0 << 3)
                                        s64 I0 := 0 + (xT0 + T0)
                                        p64 I1 := reveal s64 I0
                                        T1 := s64[I1] := 0
                                        s64 y1 := y0 + T1
                                        0 := s64[I1] := T1
                                        s64 I2 := hide p64 I1
                                        s64 0 := I2 - (xT0 + T0)
y += unsafe x[z];                       s64 0 := T0 - (z0 << 3)
```

Figure 17: Example of an unsafe array use in Hermes (Left) and RSSA (Right)

In Hermes, while unsafe arrays can be used in updates, and swaps, they cannot be used in conditions for if-else statements and for-loops since these require the variables to be public, and unsafe arrays are always secret. Therefore, no special translation is necessary for RSSA, since no Hermes program would allow this particular use of unsafe arrays.

4.7 Procedure calls and uncalls

For procedure calls and uncalls, the translation is straightforward. The syntax for RSSA contains a similar statement to Hermes' call statement, which is therefore used here. Some changes need to be applied depending on the arguments used in the call.

If an argument used is a variable, then the corresponding variable in RSSA is used. If the variable represents an array, then both variables describing the starting address of the array an its size are given. No additional statement is required in these cases.

If an argument used is an array element, then a temporary variable must be used in order to store the value of the array element. Its implementation is similar to how it is implemented in other uses of arrays. This temporary variable is therefore used as an argument in the procedure call. A similar method is used when unsafe array elements are used as arguments.

Once the arguments have been selected, the called function is executed with the defined arguments, either normally for procedure calls, or in reverse for procedure uncalls. The returning values of the function are stored in new arguments and the old ones are destroyed.

5. Tests

In order to assess the validity of our compiler, multiple tests have been made. Indeed, several Hermes programs were created in order to test the compiler. These programs are based on the example programs found in [9]. We also executed these programs using the Hermes interpreter, in order to compare the outputs from those obtained with the compiler.

The table below shows the different programs that were tested. For each, we have also presented the number of lines of the original program as well as the number of lines generated by the compiler. The original Hermes programs can be found in Section 2.1 and Appendix E.

Program	Number of Lines of Program	Number of Generated Lines
RC5 [7]	13	57
RedPike [5]	24	69
speck128 [2]	26	90
TEA [8]	15	129

Figure 23 represents the RSSA program that was generated from compiling RC5, the smallest of the generated programs we tested. The original program can be found in Figure 1.

It is clear that many more instructions have to be created in order for the original program to be accurately depicted in RSSA. The number of additional lines does not seem to depend on the number of original lines, but rather, on the original instructions and they're complexity to adapt into RSSA. For example, a simple update only requires one instruction in order to be translated into RSSA, however, a swap between a variable and an array element requires 5 different instructions.

For accurateness, using the same program with the same inputs, the outputs given by our compiler were always the same as the Hermes interpreter. We also checked by executing the programs in reverse and the output still matched between both the compiler and interpreter. Furthermore, we also compared these outputs with the results given by the RSSA interpreter, using the RSSA equivalent of the programs. In this case as well, the outputs were the same as the Hermes interpreter.

```
rc5(u64 x, u64 y[], u64 z[], u64 a[])
{
        call f(x, y, z[0], unsafe a[0]);
}
f(u64 x, u64 y[], u64 z, u64 a)
{
        x++;
        y[0]++;
        z++;
        a++;
}
```

```
begin main(s64 x0, p64 yT0, p64 yS0, p64 zT0, p64 zS0, p64 aT0, p64 aS0)
  p64 T0 := 0 ^ 0
  p64 T1 := 0 ^ (T0 << 3)
  p64 I0 := 0 ^ (zT0 + T1)
  T2 := s64[I0] := 0
  s64 T4 := 0 ^ 0
  s64 T5 := 0 ^ (T4 << 3)
  s64 I1 := 0 ^ (aT0 + T5)
  p64 I2 := reveal s64 I1
  T6 := s64[I2] := 0
  (s64 x1, p64 yT1, p64 yS1, s64 T3, s64 T7) :=
    call f(s64 x0, p64 yT0, p64 yS0, s64 T2, s64 T6)
  0 := s64[I2] := T7
  s64 I3 := hide p64 I2
  s64 0 := I3 ^ (aT0 + T5)
  s64 0 := T5 ^ (T4 << 3)
  s64 0 := T4 ^ 0
  0 := s64[I0] := T3
  p64 0 := I0 ^ (zT0 + T1)
  p64 0 := T1 ^ (T0 << 3)
  p64 0 := T0 ^ 0
end main(s64 x1, p64 yT1, p64 yS1, p64 zT0, p64 zS0, p64 aT0, p64 aS0)
begin f(s64 x0, p64 yT0, p64 yS0, s64 z0, s64 a0)
  s64 x1 := x0 + 1
  p64 T0 := 0 + (0 << 3)
  p64 I0 := 0 + (yT0 + T0)
  s64[I0] += 1
  p64 0 := I0 − (yT0 + T0)
  p64 0 := T0 − (0 << 3)
  s64 z1 := z0 + 1
  s64 a1 := a0 + 1
end f(s64 x1, p64 yT0, p64 yS0, s64 z1, s64 a1)
```

Figure 18: Example of the use of a procedure call in Hermes (Top) and RSSA (Bottom)

```
begin main(p64 ctT0, p64 ctS0, p64 ST0, p64 SS0)
  s32 A0 := 0
  s32 B0 := 0
  p64 T0 := 0 + (0 << 2)
  p64 I0 := 0 + (ctT0 + T0)
  A1 := s32[I0] := A0
  p64 0 := I0 - (ctT0 + T0)
  p64 0 := T0 - (0 << 2)
  p64 T1 := 0 + (1 << 2)
  p64 I1 := 0 + (ctT0 + T1)
  B1 := s32[I1] := B0
  p64 0 := I1 - (ctT0 + T1)
  p64 0 := T1 - (1 << 2)
  p64 T2 := 0 + (0 << 2)
  p64 I2 := 0 + (ST0 + T2)
  T3 := s32[I2] := 0
  s32 A2 := A1 + T3
  0 := s32[I2] := T3
  p64 0 := I2 - (ST0 + T2)
  p64 0 := T2 - (0 << 2)
  p64 T4 := 0 + (1 << 2)
  p64 I3 := 0 + (ST0 + T4)
  T5 := s32[I3] := 0
  s32 B2 := B1 + T5
  0 := s32[I3] := T5
  p64 0 := I3 - (ST0 + T4)
  p64 0 := T4 - (1 << 2)
  p64 i0 := 0 ^ 2
-> main0L1(p64 ctT0, p64 ctS0, p64 ST0, p64 SS0, s32 A2, s32 B2, p64 i0)
main0L1(p64 ctT1, p64 ctS1, p64 ST1, p64 SS1, s32 A3, s32 B3, p64 i1)main0L2 <- i1 == 2
  s32 A4 := A3 ^ B3
  s32 A5 := A4 << B3
  p64 T7 := 0 + (i1 << 2)
  p64 I5 := 0 + (ST1 + T7)
  T8 := s32[I5] := 0
  s32 A6 := A5 + T8
  0 := s32[I5] := T8
  p64 0 := I5 - (ST1 + T7)
  p64 0 := T7 - (i1 << 2)
  s32 B4, s32 A7 := A6, B3
  p64 i2 := i1 + 1
i2 == SS1 -> main0L3(p64 ctT1, p64 ctS1, p64 ST1, p64 SS1, s32 A7, s32 B4, p64 i2)main0L2
main0L3(p64 ctT2, p64 ctS2, p64 ST2, p64 SS2, s32 A8, s32 B5, p64 i3) <-
  p64 0 := i3 ^ SS2
  p64 T10 := 0 + (1 << 2)
  p64 I7 := 0 + (ctT2 + T10)
  B6 := s32[I7] := B5
  p64 0 := I7 - (ctT2 + T10)
  p64 0 := T10 - (1 << 2)
  p64 T11 := 0 + (0 << 2)
  p64 I8 := 0 + (ctT2 + T11)
  A9 := s32[I8] := A8
  p64 0 := I8 - (ctT2 + T11)
  p64 0 := T11 - (0 << 2)
  s32 0 := B6
  s32 0 := A9
end main(p64 ctT2, p64 ctS2, p64 ST2, p64 SS2)
```

Figure 19: The RC5 program generated into RSSA

Note that the initialization of the inputs is different between the compiler and the Hermes interpreter. For the compiler, the memory must be initialized as byte-sized, while the inputs for arrays were instead inputs for the initial address and the size of the array. For the Hermes interpreter, the array elements were sufficient in order to execute it. The differences in how to initialize the inputs in both cases is thus important to know when testing both the compiler and the Hermes interpreter.

6. Future Works

While the compiler we came up with is functional and has been able to compile many Hermes programs, it is still rather rudimentary. A lack of time on our part, means that some properties of the compiler have not been implemented. However, this could be seen as an opportunity to improve on the original compiler in order to create a more refined version of it.

Some expressions from Hermes, such as the unary operations and conditional swaps are not taken into account in the compiler. Implementing unary operations would require finding an appropriate translation to RSSA, which might require refining its extended version. Conditional swaps, on the other hand, would require a series of bitmask operations.

Local array declarations are also not implemented. These would require a pointer variable in order to keep track of the starting address in the memory. The pointer would have its value increased by the size of the given array after each declaration. This could be done by using a loop. The code below shows an example of how array declarations could look like in RSSA. A is the given array, with AT and AS representing the starting address and the size respectively, X is the given size of the array and $SMPS$ represents the pointer for the secret memory. The inverse of this code could also be used at the end of the block to free the array memory and check that it is returned to 0, thus preserving reversibility.

```
s32 A[X];
```

```
p64 AT0 = 0 + SMPS0
p64 AS0 = 0 + X << 2
p64 I0 := 0
-> initA(p64 SMPS0, p64 I0, ...)
initA(p64 SMPS1, p64 I1, ...) initA1 <- I1 == 0
0 := s32[SMPS1] := 0
p64 SMPS2 := SMPS1 + 4
p64 I2 := I1 + 1
I2 == X -> initA1(p64 SMPS2, p64 I2, ...) DoneA
DoneA(p64 SMPS3, p64 I3, ...) <-
p64 0 := I3 - X
```

Figure 20: A representation of array declarations in Hermes (Left) and RSSA (Right)

Furthermore, the compiler does not check the usage of arrays. While the user can give the starting address and the size of the array, the compiler doesn't verify if the size given fits into its memory or if given indexes fit in that size, nor does it check if a given starting address is comprised inside of an area representing another array. Arrays are thus delicate to use and users should be made aware of it before testing the compiler. Another use of pointers could be used in order for this characteristic to be implemented, increasing it for each array declared, in a similar method to local array declarations, and checking if the next given starting address is bigger than the pointer and smaller than the size of the memory used.

The incrementation of names of variables is also lacking, particularly for temporary variables, which can either be presented in the incorrect order, introducing T1 before T0 for example, skipping an incrementation, or have its incrementation reset after leaving a block, resulting in the same variable being used. While none of this results in an error for either the compiler or the RSSA interpreter, and the results given by the programs are still correct, it would be beneficial to look into this issue, in order for the translation to be less confusing. This could be done by changing when these variables are incremented, in order for the presented order to be more suitable.

Additionally, the RSSA interpreter does not take into account updates that use comparative operations like != or <, due to the rarity of these associations. While this does not impact the translation itself, it does result in an error been given during execution when such an update is generated by the compiler. A modification of the compiler is thus required in order to take comparative operations into account, likely by returning a boolean answer depending on the validity of the given expression.

Additionally, the code for the compiler and its different modules could always be improved, in order to avoid any unnecessary repetition and simplifying some of its functions.

7. Conclusions

We have shown in this paper the scheme in order to compile Hermes code into RSSA, implementing an extended version of the latter language to properly take into consideration Hermes' design. Both the RSSA interpreter and the compiler have been implemented and both can execute programs both forwards and backwards. The compiler in particular does give the same results as the Hermes interpreter, making it a satisfactory alternative from it.

While the compiler would benefit from optimizations, this first version serves as an important step in order to allow further research in reversible languages, particularly for both Hermes and RSSA.

The optimization potential of the compiler and the research is therefore vast and could lead to expanding the development of reversible languages.

Acknowledgments

We would like to thank Mr. Torben Mogensen for his supervision and involvement throughout the development of the compiler and the writing of this paper.

Appendix

A - Hermes Grammar

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Procedure}^{+} \\
 & & \\
\mathit{Procedure} & \rightarrow & \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \mathit{Stat} \\
\mathit{Args} & \rightarrow & \mathit{Type}\ \mathbf{id} \\
 & | & \mathit{Type}\ \mathbf{id}\texttt{[ ]} \\
 & | & \mathit{Args}\ \texttt{,}\ \mathit{Args} \\
\mathit{Type} & \rightarrow & \mathit{secret}\ \mathbf{IntType} \\
 & | & \mathit{public}\ \mathbf{IntType} \\
\mathit{Stat} & \rightarrow & \texttt{;} \\
 & | & \mathit{Lval}\ \mathbf{update}\ \mathit{Exp}\ \texttt{;} \\
 & | & \mathit{Lval}\ \texttt{<->}\mathit{Lval} \\
 & | & \texttt{if}\ \texttt{(}\ \mathit{Exp}\ \texttt{)}\ \mathit{Lval}\ \texttt{<->}\mathit{Lval} \\
 & | & \texttt{if}\ \texttt{(}\ \mathit{Exp}\ \texttt{)}\ \mathit{Stat}\ \texttt{else}\ \mathit{Stat}\ \texttt{;} \\
 & | & \texttt{for}\ \texttt{(}\ \mathbf{id}\ \texttt{=}\mathit{Exp}\ \texttt{;}\ \mathit{Exp}\ \texttt{)}\ \mathit{Stat} \\
 & | & \texttt{call}\ \mathbf{id}\ \texttt{(}\ \mathit{Lvals}\texttt{);} \\
 & | & \texttt{uncall}\ \mathbf{id}\ \texttt{(}\ \mathit{Lvals}\texttt{);} \\
 & | & \texttt{\{}\ \mathit{Decls}1\ \ \mathit{Stat}^{*}\texttt{\}} \\
\mathit{Exp} & \rightarrow & \mathit{Lval} \\
 & | & \mathbf{numConst} \\
 & | & \texttt{size}\ \mathbf{id} \\
 & | & \mathit{Exp}\ \mathbf{binOp}\ \mathit{Exp} \\
 & | & \mathbf{unOp}\ \mathit{Exp} \\
\mathit{Lval} & \rightarrow & \mathbf{id} \\
 & | & \mathbf{id}\ \texttt{[}\ \mathit{Exp}\ \texttt{]} \\
 & | & \texttt{unsafe}\ \mathbf{id}\ \texttt{[}\ \mathit{Exp}\ \texttt{]} \\
\mathit{Lvals} & \rightarrow & \mathit{Lval} \\
 & | & \mathit{Lval}\ \texttt{,}\ \mathit{Lvals} \\
VarSpec & \rightarrow & \mathbf{id} \\
 & | & \mathbf{id}\ \texttt{[}\ \mathit{Exp}\ \texttt{]} \\
\mathit{Decls} & \rightarrow & \\
 & | & \mathbf{type}\ VarSpecs\ \texttt{;}\ \mathit{Decls} \\
 & | & \texttt{const}\ \mathbf{type}\ \mathbf{id}\ \texttt{=}\ \mathbf{numConst}\ \texttt{;}\ \mathit{Decls}
\end{array}
$$

B - RSSA Grammar

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Block}^{+} \\
 & & \\
\mathit{Block} & \rightarrow & \mathit{Entry}\ \mathit{Stat}^{*}\ \mathit{Exit} \\
 & & \\
\mathit{Entry} & \rightarrow & \texttt{begin}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \leftarrow \\
 & | & \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \mathbf{id}\ \leftarrow\ \mathit{Cond} \\
 & & \\
\mathit{Entry} & \rightarrow & \texttt{end}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \rightarrow\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \mathit{Cond}\ \rightarrow\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \mathbf{id} \\
 & & \\
\mathit{Args} & \rightarrow & \mathit{Atom} \\
 & | & \mathit{Args}\ \texttt{,}\ \mathit{Args} \\
 & & \\
\mathit{Stat} & \rightarrow & \mathit{Atom}\ \texttt{:=}\ \mathit{Atom}\ \mathbf{updateBinOp}\ \texttt{(}\ \mathit{Atom}\ \mathbf{binOp}\ \mathit{Atom}\ \texttt{)} \\
 & | & \mathit{Atom}\ \texttt{:=}\ \mathit{Atom} \\
 & | & \mathit{Atom}\ \texttt{:=}\ \mathit{Atom}\ \mathbf{updateBinOp}\ \mathit{Atom} \\
 & | & \mathbf{id}\ \texttt{,}\ \mathbf{id}\ \texttt{:=}\ \mathbf{id}\ \texttt{,}\ \mathbf{id} \\
 & | & \mathit{Memory}\ \mathbf{updateBinOp=}\ \mathit{Atom}\ \mathbf{binOp}\ \mathit{Atom} \\
 & | & \mathit{Memory}\ \texttt{<->}\mathit{Memory} \\
 & | & \mathit{Atom}\ \texttt{:=}\ \mathit{Memory}\ \texttt{:=}\ \mathit{Atom} \\
 & | & \texttt{(}\ \mathit{Args}\ \texttt{)}\ \texttt{:=}\ \texttt{call}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \texttt{(}\ \mathit{Args}\ \texttt{)}\ \texttt{:=}\ \texttt{uncall}\ \mathbf{id}\ \texttt{(}\ \mathit{Lvals}\texttt{);} \\
 & & \\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
 & | & \mathbf{numConst} \\
 & & \\
\mathit{Memory} & \rightarrow & \texttt{M[}\ \mathit{Atom}\ \texttt{]} \\
 & & \\
\mathit{Cond} & \rightarrow & \mathit{Atom}\ \mathbf{boolBinOp}\ \mathit{Atom}
\end{array}
$$

C - Extended RSSA Grammar

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Block}^{+} \\
\\
\mathit{Block} & \rightarrow & \mathit{Entry}\ \mathit{Stat}^{*}\ \mathit{Exit} \\
\\
\mathit{Entry} & \rightarrow & \texttt{begin}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \leftarrow \\
 & | & \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \mathbf{id}\ \leftarrow\ \mathit{Cond} \\
\\
\mathit{Entry} & \rightarrow & \texttt{end}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \rightarrow\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \mathit{Cond}\ \rightarrow\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \mathbf{id} \\
\\
\mathit{Args} & \rightarrow & \mathit{Type}\ \mathit{Atom} \\
 & | & \mathit{Args}\ ,\ \mathit{Args} \\
\\
\mathit{Type} & \rightarrow & \mathbf{secretType} \\
 & | & \mathbf{publicType} \\
\\
\mathit{Stat} & \rightarrow & \mathit{Type}\ \mathit{Atom}\ \texttt{:=}\ \mathit{Atom}\ \mathbf{updateBinOp}\ \texttt{(}\ \mathit{Atom}\ \mathbf{binOp}\ \mathit{Atom}\ \texttt{)} \\
 & | & \mathit{Type}\ \mathit{Atom}\ \texttt{:=}\ \mathit{Atom} \\
 & | & \mathit{Type}\ \mathit{Atom}\ \texttt{:=}\ \mathit{Atom}\ \mathbf{updateBinOp}\ \mathit{Atom} \\
 & | & \mathit{Type}\ \mathbf{id}\ ,\ \mathit{Type}\ \mathbf{id}\ \texttt{:=}\ \mathbf{id}\ ,\ \mathbf{id} \\
 & | & \mathit{Memory}\ \mathbf{updateBinOp=}\ \mathit{Atom}\ \mathbf{binOp}\ \mathit{Atom} \\
 & | & \mathit{Memory}\ \texttt{<->}\mathit{Memory} \\
 & | & \mathit{Atom}\ \texttt{:=}\ \mathit{Memory}\ \texttt{:=}\ \mathit{Atom} \\
 & | & \texttt{(}\ \mathit{Args}\ \texttt{)}\ \texttt{:=}\ \texttt{call}\ \mathbf{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)} \\
 & | & \texttt{(}\ \mathit{Args}\ \texttt{)}\ \texttt{:=}\ \texttt{uncall}\ \mathbf{id}\ \texttt{(}\ \mathit{Lvals}\texttt{);} \\
 & | & \mathbf{publicType}\ \mathbf{id}\ \texttt{:=}\ \texttt{reveal}\ \mathbf{secretType}\ \mathbf{id} \\
 & | & \mathbf{secretType}\ \mathbf{id}\ \texttt{:=}\ \texttt{hide}\ \mathbf{publicType}\ \mathbf{id} \\
\\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
 & | & \mathbf{numConst} \\
\\
\mathit{Memory} & \rightarrow & \mathbf{Type}\ \texttt{[}\ \mathit{Atom}\ \texttt{]} \\
\\
\mathit{Cond} & \rightarrow & \mathit{Atom}\ \mathbf{boolBinOp}\ \mathit{Atom}
\end{array}
$$

D - Type Rules for extended RSSA

The following is the type rules used for the extended version of RSSA. Each is accompanied with an explanation.

Begin: The variables from the arguments a are created, thus creating the environment

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho}{\vdash_{Entry} begin(a) : [] \leftrightarrow \rho}\text{ Begin}$$

End: The variables corresponding to the arguments are passed back to the caller, putting the environment back as empty

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho}{\vdash_{Exit} end(a) : \rho \leftrightarrow []}\text{ End}$$

Conditional Entry: The variables corresponding to the arguments are created, updating the environment. The condition is verified to be true if entered through L1, and false is through L2

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho \qquad \rho \vdash_{Cond} c}{\vdash_{Entry} L1(a)L2 \leftarrow c : [] \leftrightarrow \rho}\text{ Cond. Entry}$$

Conditinal Exit: The variables corresponding to the arguments are destroyed, updating the environment. If the condition is true, a jump to L1 is made, else a jump to L2 is made.

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho \qquad \rho \vdash_{Cond} c}{\vdash_{Exit} c \rightarrow L1(a)L2 : \rho \leftrightarrow []}\text{ Cond. Exit}$$

Unconditional Entry: The variables corresponding to the arguments are created, updating the environment.

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho}{\vdash_{Entry} L(a) \leftarrow: [] \leftrightarrow \rho}\text{ Uncond. Entry}$$

Unconditinal Exit: The variables corresponding to the arguments are destroyed, updating the environment.

$$\frac{\vdash_{Args} a : [] \leftrightarrow \rho}{\vdash_{Exit}\rightarrow L(a) : \rho \leftrightarrow []}\text{ Uncond. Exit}$$

Use Constant: The rule is used when an atom is used in a statement where it is no removed from the environment. An atom that is a constant is assigned the type p64, representing a public type of 64 bits.

$$\frac{}{\rho \vdash_{Use} n : p64}\text{ Use Const.}$$

Use: Similar to the previous rule only that the type is assigned.

$$\frac{x \in dom(\rho)}{\rho \vdash_{Use} x : t}\text{ Use}$$

Atom Constant: An atom that is a constant is assigned a type p64.

$$\frac{}{\rho \vdash_{Atom} n : p64}\text{ Atom Const.}$$

Atom: An atom that is a variable is assigned a type. The name of the variable must be different from any other variable in the environment's domain.

$$\frac{x \notin dom(\rho)}{\vdash_{Atom} t\ x : \rho \leftarrow t \rightarrow \rho[x \mapsto t]}\text{ Atom}$$

Arguments Constant: An argument declared as a constant does not have any effect on the environment, thus it stays the same.

$$\frac{}{\rho \vdash_{Args} n \rightarrow \rho}\text{ Arguments Const.}$$

Single Argument: Given an argument x and its type t, the environment is updated in order to include the x, assigned with type t. An argument cannot have the same name as any variables in the environment's domain.

$$\frac{x \notin dom(\rho)}{\vdash_{Args} t\ x : \rho \leftrightarrow \rho[x \mapsto t]}\text{ Single Argument}$$

Arguments: When multiple arguments are given, the environment is updated, adding all arguments to it.

$$\frac{\vdash_{Args} a_1 : \rho \leftrightarrow \rho' \qquad \vdash_{Args} a_s : \rho' \leftrightarrow \rho''}{\vdash_{Args} a_1, a_s : \rho \leftrightarrow \rho''}\text{ Arguments}$$

Memory Access: The atom must be of type p64 in order to be used as an index to the memory. The memory is bound by a type

$$\frac{\rho \vdash_{Use} a : p64}{\rho \vdash_{Memory} t[a] \rightarrow t}\text{ Memory Access}$$

Condition: Both atoms must have a public type for the condition to be made.

$$\frac{\rho \vdash_{Use} a_1 : p \qquad \rho \vdash_{Use} a_2 : p}{\rho \vdash_{Cond} a_1 \bowtie a_2}\text{ Condition}$$

Single Assignment 1: The variable a_1 is created through the calculation $a_2 \oplus (a_3 \odot a_4)$. It represents an update, going from variable a_2, to variable a_1. For this, a_1 and a_2 must have the same type. a_2 is also eliminated in the environment. a_3 and a_4 must be different from both a_1 and a_2. If either one of their types is secret, t_1 must also be secret.

$$\frac{\vdash_{Atom} a_2 : \rho' \leftarrow t_1 \rightarrow \rho \qquad \vdash_{Atom} a_1 : \rho' \leftarrow t_1 \rightarrow \rho'' \qquad \rho \vdash_{Use} a_3 : t_2 \qquad \rho \vdash_{Use} a_4 : t_3}{\Gamma \vdash_{Stat} a_1 := a_2 \oplus (a_3 \odot a_4) : \rho \leftrightarrow \rho''}\text{ Single Assign. 1}$$

Single Assignment 2: The value of a_2 is given to a_1. Thus a_1 is added to the environment and a_2 is removed from it.

$$\frac{\vdash_{\mathit{Atom}} a_2 : \rho' \leftarrow t_1 \rightarrow \rho \qquad \vdash_{\mathit{Atom}} a_1 : \rho' \leftarrow t_1 \rightarrow \rho''}{\Gamma \vdash_{\mathit{Stat}} a_1 := a_2 : \rho \leftrightarrow \rho''} \text{ Single Assign. 2}$$

Double Assignment: Two previous variables are used to created two new variables simultaneously. The previous variables are destroyed.

$$\frac{\vdash_{\mathit{Atom}} a_2 : \rho' \leftarrow t_1 \rightarrow \rho \qquad \vdash_{\mathit{Atom}} a_3 : \rho'' \leftarrow t_2 \rightarrow \rho' \qquad \vdash_{\mathit{Atom}} a_0 : \rho'' \leftarrow t_1 \rightarrow \rho''' \qquad \vdash_{\mathit{Atom}} a_1 : \rho''' \leftarrow t_2 \rightarrow \rho''''}{\Gamma \vdash_{\mathit{Stat}} a_0, a_1 := a_2, a_3 : \rho \leftrightarrow \rho''''} \text{ Double Assign.}$$

Memory Update: The memory is updated using the atoms a_1 and a_2. Both atoms have the same type, and if they are secret, than the type of the memory must also be secret. If it is public, the memory can have any kind of type, public or secret. The environment stays the same.

$$\frac{\rho \vdash_{\mathit{Memory}} m \rightarrow s \qquad \rho \vdash_{\mathit{Use}} a_1 : t \qquad \rho \vdash_{\mathit{Use}} a_2 : t}{\Gamma \vdash_{\mathit{Stat}} m\oplus = a_1 \odot a_2 : \rho \leftrightarrow \rho} \text{ Memory Update 1}$$

$$\frac{\rho \vdash_{\mathit{Memory}} m \rightarrow p \qquad \rho \vdash_{\mathit{Use}} a_1 : p \qquad \rho \vdash_{\mathit{Use}} a_2 : p}{\Gamma \vdash_{\mathit{Stat}} m\oplus = a_1 \odot a_2 : \rho \leftrightarrow \rho} \text{ Memory Update 2}$$

Memory Swap: Both entries of the memory are swapped. The environment stays the same.

$$\frac{\rho \vdash_{\mathit{Memory}} m_1 \rightarrow t \qquad \rho \vdash_{\mathit{Memory}} m_2 \rightarrow t}{\Gamma \vdash_{\mathit{Stat}} m_1 \leftrightarrow m_2 : \rho \leftrightarrow \rho} \text{ Memory Swap}$$

Variable-Memory Swap: The value of the memory access is given to the atom a_1 which is added to the environment. The memory access gets the value of the old atom a_2 which is then eliminated. The address variable used in the memory must be different than both a_1 and a_2.

$$\frac{\vdash_{\mathit{Atom}} a_2 : \rho' \leftarrow t \rightarrow \rho \qquad \rho' \vdash_{\mathit{Memory}} m \rightarrow t \qquad \vdash_{\mathit{Atom}} a_1 : \rho' \leftarrow t \rightarrow \rho''}{\Gamma \vdash_{\mathit{Stat}} a_1 := m := a_2 : \rho \leftrightarrow \rho''} \text{ Variable-Memory Swap}$$

Call: The function f is called with the arguments a_2 and its results are given to new arguments. We also check if the number of arguments for a_2 are the same as the number of inputs in f, and if they have similar types. The arguments a_1 created are added to the environment. We also check if those arguments have the same number and type as the output of f. The arguments a_2 used in the call to the function f are eliminated.

$$\frac{\vdash_{\mathit{Args}} a_2 : \rho' \leftarrow t_1^* \rightarrow \rho \qquad \vdash_{\mathit{Args}} a_1 : \rho' \leftarrow t_2^* \rightarrow \rho'' \qquad \Gamma(f, \mathit{true}) \rightarrow t_1^* \qquad \Gamma(f, \mathit{false}) \rightarrow t_2^*}{\Gamma \vdash_{\mathit{Stat}} a_1 := \mathit{call} f(a_2) : \rho \leftrightarrow \rho''} \text{ Call}$$

Uncall: Similar to uncall only doing the reverse function of f.

$$\frac{\Gamma \vdash_{Stat} a_2 := call f(a_1) : \rho' \leftrightarrow \rho}{\Gamma \vdash_{Stat} a_1 := uncall f(a_2) : \rho \leftrightarrow \rho'}\text{Uncall}$$

Reveal: An atom of type secret has its value given to a new atom of type public. Both atoms must have the same size.

$$\frac{\vdash_{Atom} a_2 : \rho' \leftarrow sZ \rightarrow \rho \qquad \vdash_{Atom} a_1 : \rho' \leftarrow pZ \rightarrow \rho''}{\Gamma \vdash_{Stat} a_1 := reveal\ a_2 : \rho \leftrightarrow \rho''}\text{Reveal}$$

Hide: An atom of type public has its value given to a new atom of type secret. Both atoms must have the same size.

$$\frac{\vdash_{Atom} a_2 : \rho' \leftarrow pZ \rightarrow \rho \qquad \vdash_{Atom} a_1 : \rho' \leftarrow sZ \rightarrow \rho''}{\Gamma \vdash_{Stat} a_1 := hide\ a_2 : \rho \leftrightarrow \rho''}\text{Hide}$$

E - Hermes test code

The following are some of the Hermes programs we used for testing our compiler.

```
encrypt(u32 x[], u32 k[])
{
  const CONST = 0x9E3779B9;
  u32 rk0, rk1, x0, x1;

  {
    rk0 <-> k[0]; rk1 <-> k[1];
  } @ {
    x0 <-> x[0]; x1 <-> x[1];
    for (i = 0; 16) {
      rk0 += CONST;
      rk1 -= CONST;
      x0 ^= rk0;
      x0 += x1;
      x0 <<= x1;
      x1 >>= x0;
      x1 -= x0;
      x1 ^= rk1;
      i++;
    }
    rk0 -= CONST<<4; rk1 += CONST<<4;
  }
  x0 <-> x[1]; x1 <-> x[0];
}
```

Figure 21: The RedPike program in Hermes

```
speck128(u64 ct[], u64 K[])
{
   u64 y, x, b, a;
   { y <-> ct[0]; x <-> ct[1]; b += K[0]; a += K[1]; } @
   {
     call Rs(x, y, b);
     for (i=0; 31) {
       call Rp(a, b, i);
       call Rs(x, y, b);
       i++;
     }
     for (i=31; 0) {  /* restore a and b */
       i--;
       uncall Rp(a, b, i);
     }
   }
}

/* two variants of R needed because one is called with secret k
   and another with public k */

Rs(u64 x, u64 y, secret u64 k)
{ x >>= 8; x += y; x ^= k; y <<= 3; y ^= x; }

Rp(u64 x, u64 y, public u64 k)
{ x >>= 8; x += y; x ^= k; y <<= 3; y ^= x; }
```

Figure 22: The speck128 program in Hermes

```
encrypt (u32 v[], u32 k[])
{
    u32 v0, v1, k0, k1, k2, k3;
    public u32 sum;
    const delta = 0x9E3779B9;
/* key schedule constant */
    v0 <-> v[0]; @ v1 <-> v[1]; @
/* set up */
    k0 += k[0]; @ k1 += k[1]; @ k2 += k[2]; @ k3 += k[3]; @ /* cache key */
    for (i=0; 32) {
/* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        i++;
    }
/* end cycle */
    sum -= delta << 5;    /* alternatively, sum -= 0xC6EF3720 */
}
```

Figure 23: The TEA program in Hermes

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 1–11, New York, NY, USA, 1988. Association for Computing Machinery.

[2] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Paper 2013/404, 2013. https://eprint.iacr.org/2013/404.

[3] Martin Kutrib, Uwe Meyer, Niklas Deworetzki, and Marc Schuster. *Compiling Janus to RSSA*, pages 64–78. 06 2021.

[4] Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

[5] C Mitchell, S Murphy, F Piper, and P Wild. Red pike—an assessment. *Codes and Ciphers Ltd*, 2:10–96, 1996.

[6] Torben Ægidius Mogensen. RSSA: A reversible SSA form. In Manuel Mazzara and Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 203–217. Springer International Publishing, 2016.

[7] Ronald L. Rivest. The rc5 encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, pages 86–96, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[8] David J. Wheeler and Roger M. Needham. Tea, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, pages 363–366, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[9] Torben Ægidius Mogensen. Hermes: A reversible language for lightweight encryption. *Science of Computer Programming*, 215:102746, 2022.