Hermes: A Language for Light-Weight Encryption

Torben Ægidius Mogensen(✉)

DIKU, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark
torbenm@di.ku.dk

Abstract. Hermes is a domain-specific language for writing light-weight encryption algorithms: It is reversible, so it is not necessary to write separate encryption and decryption procedures, and it avoids several types of side-channel attacks, both by ensuring no secret values are left in memory and by ensuring that operations on secret data spend time independent of the value of this data, thus preventing timing-based attacks. We show a complete formal specification of Hermes, argue absence of timing-based attacks (under reasonable assumptions), and compare implementations of well-known light-weight encryption algorithms in Hermes and C.

1 Introduction

Recent work [12] have investigated using the reversible language Janus [5,19] for writing encryption algorithms. Janus is a structured imperative language where all statements are reversible. A requirement for reversibility is that no information is ever discarded: No variable is destructively overwritten in such a way that the original value is lost. Instead, it must be updated in a reversible manner or swapped with another variable. Since encryption is by nature reversible, it seems natural to write these in a reversible programming language. Additionally, reversible languages requires that all intermediate variables are cleared to 0 before they are discarded, which ensures that no information that could potentially be used for side-channel attacks is left in memory. But non-cleared variables is not the only side-channel attack used against encryption: If the time used to encrypt data can depend on the values of the data and the encryption key, attackers can gain (some) information about the data or the key simply by measuring the time used for encryption. Janus has control structures the timing of which depend on the values of variables, so it does not protect against timing-based attacks.

So we propose a new reversible language, Hermes, specifically designed to address these concerns. Although somewhat inspired by Janus, Hermes has some significant differences, as we shall see below. An early version of the Hermes language was presented in [7]. Experiments using this language have indicated a need for a type system that separates secret and public data. In the early version, the (informally specified) type system distinguishes constants, loop variables, and all else, with constants and loop variables being considered non-secret and

I. Lanese and M. Rawski (Eds.): RC 2020, LNCS 12227, pp. 93–110, 2020.
https://doi.org/10.1007/978-3-030-52482-1_5

all else being secret. This early language is, however, too restrictive in many cases and too permissive in other cases:

- Loop bounds and array sizes were constants, so algorithms with variable-size keys or data would have to have a procedure for each size.
- Loop counters could in the early version of Hermes only be updated by constant values, which may also be too restrictive.
- Procedure parameters are not distinguished by secrecy, so loop counters could not be passed as parameters. By classifying parameters as public or secret, loop counters can now be passed as public parameters.
- Any value was allowed as index to an array, but since timing can depend on the index value (due to caching), this is a potential side channel. By limiting array indices to public values, this can be avoided.

So we propose a new version of Hermes that uses public and secret types, with strong restrictions on operations on secret values. Constants and loop counters are public, all other variables are by default secret, but can be declared public. The type system not only tracks flow of information similar to binding-time analysis [3], trust analysis [8], and information flow analysis [11] but also imposes restrictions to ensure reversibility and (under reasonable assumptions) avoid timing-based side-channel attacks.

Program → *Procedure*$^+$

Procedure → **id** (*Args*) *Stat*

Args → *Type* **id** | *Type* **id**`[]` | *Args* , *Args*

Type → `secret` **IntType** | `public` **IntType**

Stat → ;
| *Lval* **update** *Exp* ;
| *Lval* `<->` *Lval* | `if` (*Exp*) *Lval* `<->` *Lval*
| `for` (**id** =*Exp* ; *Exp*) *Stat*
| `call` **id** (*Lvals*); | `uncall` **id** (*Lvals*);
| { *Decls*1 *Stat**}

Exp → *Lval* | **numConst** | `size` **id**
| *Exp* **binOp** *Exp* | **unOp** *Exp*

Lval → **id** | **id** [*Exp*]

Lvals → *Lval* | *Lval* , *Lvals*

VarSpec → **id** | **id** [*Exp*]

Decls →
| *Type VarSpec* ; *Decls*
| `const` **id** = **numConst** ; *Decls*

Fig. 1. Core syntax of Hermes

2 Hermes Syntax

The core syntax of Hermes is shown in Fig. 1. The grammar uses tokens specified in boldface. These are described below.

id denotes identifiers. An identifier starts with a letter and can contain letters, digits, and underscores.
numConst denotes decimal or hexadecimal integers using C-style notation.
IntType denotes names of integer types. These can be `u8`, `u16`, `u32`, and `u64`, representing unsigned integers of 8, 16, 32 or 64 bits.
unOp denotes an unary operator on numbers. This can be bitwise negation (`~`).
binOp denotes an unary operator on numbers. This can be one of `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `<<`, and `>>`. All arithmetic is modulo 2^{64}. Comparison operators return $2^{64}-1$ (all ones) when the comparison is true and 0 when the comparison is false. Note that this is different from their behaviour in C, where they return 1 and 0, respectively. `&`, `|`, and `^` are bitwise logical operators.
update denotes an update operator. This can be one of `+=`, `-=`, `^=`, `<<=`, and `>>=`. The first three operators have the same meaning as in C. `<<=` is a left rotate. The rotation amount is modulo the size of the L-value being rotated, so if, for example, `x` is an 8-bit variable, `x <<= 13;` will rotate `x` left by 5 bits. `>>=` is a right rotate using similar rules. Note that the meaning of `<<=` and `>>=` differ from their meaning in C, where they represent shift-updates.

3 The Type System of Hermes

Values in Hermes are all 64 bit unsigned integers, and they can be secret or public. Scalar and array variables additionally impose a number size (8, 16, 32 or 64 bits). A constant just has the type `constant`, which is implicitly a 64-bit number. So we have:

$$\begin{array}{lll} ValType & \rightarrow \texttt{secret} \mid \texttt{public} & \\ VarType & \rightarrow \texttt{constant} \mid ValType^{Size} \mid ValType^{Size}\texttt{[]} & \\ Size & \rightarrow 8 \mid 16 \mid 32 \mid 64 & \end{array}$$

We use t with optional subscript to denote a value type, τ with optional subscript to denote a variable type, and z with optional subscript to denote a size. So t^z denotes the special case of variable types where the variable is a scalar non-constant. We define a partial order $\sqsubseteq$ as the reflexive extension of `public` $\sqsubset$ `secret` and a least upper bound operator $\sqcup$ induced by this partial order. We use this to make the result secret when secret and public values are mixed.

3.1 L-Values and Expressions

Variable environments, denoted by ρ with optional subscript, bind identifiers (denoted by x with optional subscript) to variable types. Environments are functions, so $\rho(x)$ is the variable type that x is bound to in ρ. We update environments

using the notation $\rho[x \mapsto \tau]$, which creates a new environment that is identical to ρ, except that x is bound to τ.

Sequents for typing expressions, denoted by e with optional subscript, are of the form $\rho \vdash_E e : ValType$, and sequents for typing L-values (denoted by l with optional subscript) are of the form $\rho \vdash_L l : VarType$. In order to make updates, swaps, and parameter passing reversible, we must impose restrictions to avoid aliasing and similar clashes. To do this, we introduce functions that find variables in expressions or parts of expressions. $V()$ finds the variables in an expression or L-value, $R()$ finds the root variable of an L-value, and $V()_I$ finds the variables in index expressions in an L-value.

$$\begin{array}{lll} V(n) & = \emptyset & \qquad V(x) = \{x\} \\ V(x[e]) & = \{x\} \cup V(e) & \qquad V(\neg e) = V(e) \\ V(e_1 \odot e_2) & = V(e_1) \cup V(e_2) & \qquad V(\texttt{size}\ x) = \emptyset \end{array}$$

$$\begin{array}{lll} R(x) & = x & \qquad V_I(x) = \emptyset \\ R(x[e]) & = x & \qquad V_I(x[e]) = V(e) \end{array}$$

Note that $V()$ does not include variables in `size`-expressions, as these are harmless in terms of aliasing.

We specify rules for L-values and expressions in Fig. 2.

For L-values, the rule for variables says that a variable has the type specified by the environment. The rule for array access says that the array variable must have an array type and the index expression must be public. This ensures that timing of memory accesses (which can depend on the address, but not the accessed value) does not leak secret information. The rules for constants state that a constant is public. n denotes an integer constant. The rule for non-constant L-values say that the L-value must be a scalar and that the expression type is the value type part of the type of the L-value. The rule for an unary operator $\neg$ just say that the result has the same type as its argument. The rules for a binary operator $\odot$ is more complex. If any of the arguments are secret, the result is also secret. Additionally, some potentially time-variant operations are not allowed on secret values. We assume a set TV of time-variant operators is given. This will typically contain division and modulo operators, but can also contain multiplication if the target architecture does not have a constant-time multiplication instruction. The last rule states that the size of an array is a public value.

3.2 Statements and Local Declarations

A seqent for a statement s is of the form $\Gamma, \rho \vdash_S s$ and states that given a procedure environment Γ and variable environment ρ, the statement s is well typed. A procedure environment binds procedure names to lists of variable types. The type rules for statements are shown in Fig. 3.

The first rule says that the empty statement is well typed. To ensure reversibility, the rule for updates (where $\oplus=$ denotes an update operator) says

$$\frac{}{\rho \vdash_L x : \rho(x)}(\text{Variable}) \qquad \frac{\rho(x) = t^z\,[\,] \quad \rho \vdash_E e : \texttt{public}}{\rho \vdash_L x[e] : t^z}(\text{ArrayAccess})$$

$$\frac{}{\rho \vdash_E n : \texttt{public}}(\text{Constant1}) \qquad \frac{\rho \vdash_L l : \texttt{constant}}{\rho \vdash_E l : \texttt{public}}(\text{Constant2})$$

$$\frac{\rho \vdash_L l : t^z}{\rho \vdash_E l : t}(\text{L-val}) \qquad \frac{\rho \vdash_E e : t}{\rho \vdash_E \neg e : t}(\text{UnOp})$$

$$\frac{\rho \vdash_E e_1 : t_1 \quad \rho \vdash_E e_2 : t_2 \quad t_1 \sqcup t_2 = \texttt{public}}{\rho \vdash_E e_1 \odot e_2 : \texttt{public}}(\text{BinOp1})$$

$$\frac{\rho \vdash_E e_1 : t_1 \quad \rho \vdash_E e_2 : t_2 \quad t_1 \sqcup t_2 = \texttt{secret} \quad \odot \notin TV}{\rho \vdash_E e_1 \odot e_2 : \texttt{secret}}(\text{BinOp2})$$

$$\frac{\rho(x) = t^z\,[\,]}{\rho \vdash_E \texttt{size}\, x : \texttt{public}}(\text{Size})$$

Fig. 2. Type rules for L-values and expressions

that the root variable of the L-val must not occur in the expression. Furthermore, if the expression is secret, the L-Val must also be secret. The rule for a swap states that the two L-values must have exactly the same type, and that the root variable of one side can not occur in index expressions on the other side. The rule for conditional swap additionally requires that the root variables of the L-values do not occur in the condition and that the condition is no more secret than the L-values. The rule for loops state that the loop bounds must be public, and that the loop variable is implicitly declared to be a public 64-bit variable local to the loop body. The rules for procedure calls state that the types of the argument L-values must match those found in the procedure environment. Furthermore, to avoid aliasing and ensure reversibility, the root variable of one argument can not occur in another argument. The rule for blocks states that all statements in the block must be well typed in the environment that is extended by the local declarations. Static scoping is used. The bottom of Fig. 3 show the rules for extending environments.

Sequents for declarations are of the form $\rho \vdash_D d \rightsquigarrow \rho_1$, and state that the declaration d extends the environment ρ to ρ_1. The first rule state that an empty declaration does not change the environment. The rule for constant declarations extends the environment with the constant name bound to `constant`. The rules for variable declarations are straightforward. The rules for array declarations require that the expression that determines the size of an array must be public, and that the array variable can not shadow any variable used in this expression.

3.3 Procedures and Programs

The rules for declarations of procedures and programs are shown in Fig. 4. A sequent of the form $\vdash pgm$ states that pgm is a valid program. $\vdash_P p \rightsquigarrow \Gamma$ states

$$\frac{}{\Gamma, \rho \vdash_S \texttt{;}}(\text{Empty})$$

$$\frac{\rho \vdash_L l : t_0^z \quad \rho \vdash_E e : t_1 \quad R(l) \notin V(e) \quad t_1 \sqsubseteq t_0}{\Gamma, \rho \vdash_S l \oplus\!= e}(\text{Update})$$

$$\frac{\rho \vdash_L l_1 : t^z \quad \rho \vdash_L l_2 : t^z \quad R(l_1) \notin V_I(l_2) \quad R(l_2) \notin V_I(l_1)}{\Gamma, \rho \vdash_S l_1 \texttt{<->} l_2}(\text{Swap})$$

$$\frac{\begin{array}{c}\rho \vdash_L l_1 : t_0^z \quad \rho \vdash_L l_2 : t_0^z \quad \rho \vdash_E e : t_1 \quad t_1 \sqsubseteq t_0 \\ R(l_1) \notin V_I(l_2) \cup V(e) \quad R(l_2) \notin V_I(l_1) \cup V(e)\end{array}}{\Gamma, \rho \vdash_S \texttt{if (}e\texttt{)}\ l_1 \texttt{<->} l_2}(\text{SwapC})$$

$$\frac{\rho \vdash_E e_1 : \texttt{public} \quad \rho \vdash_E e_2 : \texttt{public} \quad \Gamma, \rho[x \mapsto \texttt{public}^{64}] \vdash_S s}{\Gamma, \rho \vdash_S \texttt{for (}x \texttt{=} e_1\texttt{;}\ e_2\texttt{)}\ s}(\text{ForLoop})$$

$$\frac{\begin{array}{c}\Gamma(f) = (\tau_1, \ldots, \tau_n) \quad \forall i \in [1,n] : \rho \vdash_L l_i : \tau_i \\ \forall i, j \in [1,n] : i \neq j \Rightarrow R(l_i) \notin V(l_j)\end{array}}{\Gamma, \rho \vdash_S \texttt{call}\ f\texttt{(}l_1, \ldots, l_n\texttt{);}}(\text{Call})$$

$$\frac{\Gamma, \rho \vdash_S \texttt{call}\ f\texttt{(}l_1, \ldots, l_n\texttt{);}}{\Gamma, \rho \vdash_S \texttt{uncall}\ f\texttt{(}l_1, \ldots, l_n\texttt{);}}(\text{Uncall})$$

$$\frac{\rho \vdash_D d \rightsquigarrow \rho_1 \quad \forall i \in [1,n] : \Gamma, \rho_1 \vdash_S s_i}{\Gamma, \rho \vdash_S \{d\ s_1 \ldots s_n\}}(\text{Block})$$

$$\frac{}{\rho \vdash_D \quad \rightsquigarrow \rho}(\text{EmptyDecl}) \qquad \frac{\rho[x \mapsto t^z] \vdash_D d \rightsquigarrow \rho_1}{\rho \vdash_D t\ \texttt{uz}\ x\texttt{;}\ \ d \rightsquigarrow \rho_1}(\text{VarDecl})$$

$$\frac{\rho[x \mapsto \texttt{constant}] \vdash_D d \rightsquigarrow \rho_1}{\rho \vdash_D \texttt{const}\ x \texttt{=} n\texttt{;}\ \ d \rightsquigarrow \rho_1}(\text{ConstDecl})$$

$$\frac{\rho \vdash_E e : \texttt{public} \quad x \notin V(e) \quad \rho[x \mapsto t^z\,\texttt{[]}] \vdash_D d \rightsquigarrow \rho_1}{\rho \vdash_D t\ \texttt{uz}\ x[e]\texttt{;}\ \ d \rightsquigarrow \rho_1}(\text{ArrayDecl})$$

Fig. 3. Type rules for statements and declarations

that a procedure p generates a procedure environment Γ, $\Gamma \vdash^P p$ states that, given the procedure environment Γ, the procedure p is valid, and $\vdash_A a \rightsquigarrow V/\overline{\tau}$ states that the argument list a generates the variable list V and the type list $\overline{\tau}$. We use $\uplus$ to append two (variable or type) lists and $\cap$ to represent the set of elements common to two lists.

The rule for programs first builds a procedure environment, ensuring that no procedure is declared twice, and then checks that all procedures are well typed in this procedure environment. Procedures can all call each other. The *Procedure*1 rule builds a procedure environment for a single procedure, and *Procedure*2 checks that a single procedure is well typed. Both use rules for building a list of argument names and types, ensuring no name occurs twice.

$$\frac{\forall i \in [1,n] : \vdash_P p_i \rightsquigarrow [f_i \mapsto (\overline{\tau_i})] \qquad \forall i,j \in [1,n] : i \neq j \Rightarrow f_i \neq f_j \qquad \forall i \in [1,n] : [f_1 \mapsto (\overline{\tau_1}), \ldots, f_n \mapsto (\overline{\tau_n})] \vdash^P p_i}{\vdash p_1 \ldots p_n}(\text{Program})$$

$$\frac{\vdash_A a \rightsquigarrow V/\overline{\tau}}{\vdash_P f\ (a)\ s \rightsquigarrow [f \mapsto \overline{\tau}]}(\text{Procedure1})$$

$$\frac{\vdash_A a \rightsquigarrow [x_1, \ldots, x_n]/[\tau_1, \ldots, \tau_n] \qquad \Gamma, [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash_S s}{\Gamma \vdash^P f\ (a)\ s}(\text{Procedure2})$$

$$\frac{\vdash_A a_1 \rightsquigarrow V_1/\overline{\tau_1} \qquad \vdash_A a_2 \rightsquigarrow V_2/\overline{\tau_2} \qquad V_1 \cap V_2 = \emptyset}{\vdash_A a_1\ ,\ a_2 \rightsquigarrow V_1 \uplus V_2/\overline{\tau_1} \uplus \overline{\tau_2}}(\text{ArgList})$$

$$\frac{}{\vdash_A t\ \ \mathtt{u}z\ x \rightsquigarrow [x]/[t^z]}(\text{Scalar}) \qquad \frac{}{\vdash_A t\ \ \mathtt{u}z\ x\,[\,] \rightsquigarrow [x]/[t^z\,[\,]]}(\text{Array})$$

Fig. 4. Type rules for procedures and programs

4 Run-Time Semantics of Hermes

The run-time semantics of Hermes does not distinguish secret and public values – type checking ensures that no secrets leak into public variables – so values in Hermes are just sized numbers. Expressions all evaluate to 64 bit numbers, which are only truncated when used to update variables or array elements, which can be 8, 16, 32, or 64 bits in size. An array has an element size, a vector size, and a vector of elements of the vector size. The sizes of scalar variables and the element sizes of array are known at compile time, but for specification convenience they are part of the run-time environments. A compiler can check sizes at compile time, so the run-time environments bind names (or offsets) to locations only. Similarly, named constants can be eliminated at compile time, so they do not need to be part of the run-time environments.

Environments (η) bind constants to their value and variables to their integer sizes (8, 16, 32, or 64) and locations.

Stores (σ) bind locations to values. The value of a scalar variable is an 8, 16, 32, or 64 bit integer, and the value of an array is a record (struct) of its vector size and its vector. The elements of the vector are locations holding 8, 16, 32, or 64 bit integers, according to the integer size of the array.

We use the same notation for environments as in the type semantics, but we also use the update notation as a pattern: If η_1 is known, we use the notation $\eta_2[x \mapsto v] = \eta_1$ to say that η_2 is equal to η_1 with the *latest* binding of x removed. This means that earlier bindings of x are retained in the environment and can be retrieved. The environments are stack-like: Bindings are removed in the opposite order in which they are created. Stores, on the other hand, do not need to retain older bindings of locations, so when a new value is bound to a location, the old value can be forgotten. We use the notation $\sigma[\lambda := v]$ when updating stores. While this is not immediately evident from the semantic rules, there is only

be one store in use at any given time, and locations are disposed of in the opposite order of their creation, so the store acts like and can be implemented as a global stack, allocating new zero-initialised locations on the top of the stack and removing them in the opposite order of their allocation.

We use a family of functions $newlocation_z$ where z an integer size (8, 16, 32, or 64) that takes a store σ returns a new store σ_1 and location λ of size z such that λ is bound to zero in σ_1, and the dual function $disposelocation_z$ that takes a storeσ_1 and a location λ and returns a store σ obtained by removing (unstacking) λ from σ_1, after checking that the contents of λ in σ_1 is 0. If not, the result is undefined. If $(\sigma_1, \lambda) = newlocation_z(\sigma)$, then $\sigma = disposelocation_z(\sigma_1\lambda)$.

We also use a family of functions $newarray_z$ that each take a store σ and a vector size vs and returns a new store σ_1 and a location λ that in the new store is bound to two fields: $\sigma_1(\lambda) = (vs, ve)$, where vs is the vector size at this location, and ve is a vector of new locations for the elements of the vector, all of which are bound to zero in the new store. We use array notation to access elements of a vector. $newarray_z$ also have duals, $disposearray_z$, that each take a store σ_1, a vector size vs, and a location λ and returns a new store σ where the array at λ has been removed (unstacked). It checks that the vector size at the location matches vs, and that all vector elements are locations with zero as content. If either of these is not true, the result is undefined. If $(\sigma_1, \lambda) = newarray_z(\sigma, vs)$, then $\sigma = disposearray_z(\sigma_1, vs, \lambda)$.

$$\frac{}{\sigma, \eta \models_L x \mathbin{@} \eta(x)}(\text{Variable/Constant})$$

$$\frac{\eta(x) = (z, \lambda) \quad \sigma(\lambda) = (vs, ve) \quad \sigma, \eta \models_E e \to i \quad i < vs}{\sigma, \eta \models_L x[e] \mathbin{@} (z, ve[i])}(\text{ArrayElement})$$

$$\frac{}{\sigma, \eta \models_E n \to n}(\text{Constant1}) \qquad \frac{\eta(x) = (n, \mathsf{null})}{\sigma, \eta \models_E x \to n}(\text{Constant2})$$

$$\frac{\sigma, \eta \models_L l \mathbin{@} (z, \lambda)}{\sigma, \eta \models_E l \to \sigma(\lambda) \uparrow_z}(\text{L-val}) \qquad \frac{\sigma, \eta \models_E e \to v}{\sigma, \eta \models_E \neg e \to I(\neg)(v)}(\text{UnOp})$$

$$\frac{\sigma, \eta \models_E e_1 \to v_1 \quad \sigma, \eta \models_E e_2 \to v_2}{\sigma, \eta \models_E e_1 \odot e_2 \to I(\odot)(v_1, v_2)}(\text{BinOp}) \qquad \frac{\eta(x) = (z, \lambda) \quad \sigma(\lambda) = (vs, ve)}{\sigma, \eta \models_E \mathtt{size}\, x \to vs}(\text{Size})$$

Fig. 5. Semantic rules for L-values and expressions

4.1 L-Values and Expressions

Figure 5 shows the evaluation rules for L-values and expressions. L-values evaluate to locations, and expressions to 64-bit integers. Sequents for L-values are of the form $\sigma, \eta \models_L l \mathbin{@} (z, \lambda)$ and state that the L-value l is stored at location λ which is of size z. We use a special case for constants: When $\lambda = \mathsf{null}$, l is a constant equal to z. null is a null location where no values are stored.

$$
\begin{array}{rcl@{\qquad}rcl}
I(;) & = & ; & I(l \texttt{ ^= } e;) & = & l \texttt{ ^= } e; \\
I(l \texttt{ += } e;) & = & l \texttt{ -= } e; & I(l \texttt{ -= } e;) & = & l \texttt{ += } e; \\
I(l \texttt{ <<= } e;) & = & l \texttt{ >>= } e; & I(l \texttt{ >>= } e;) & = & l \texttt{ <<= } e;
\end{array}
$$

$$
I(l_1 \texttt{ <-> } l_2;) = l_1 \texttt{ <-> } l_2;
$$

$$
I(\texttt{if } (c)\ l_1 \texttt{ <-> } l_2;) = \texttt{if } (c)\ l_1 \texttt{ <-> } l_2;
$$

$$
I(\texttt{for } (x \texttt{ = } e_1; e_2)\ s) = \texttt{for } (x \texttt{ = } e_2; e_1)\ I(s)
$$

$$
I(\texttt{call } f(as);) = \texttt{uncall } f(as); \qquad I(\texttt{uncall } f(as);) = \texttt{call } f(as);
$$

$$
I(\{d\ s_1 \ldots s_n\}) = \{d\ I(s_n) \ldots I(s_1)\}
$$

Fig. 6. Inverting statements

Sequents of the form $\sigma, \eta \models_E e \rightarrow v$, state that e evaluates to v.

We use a function I that binds operator symbols to the functions they represent. So $I(\texttt{+})$ is a function that takes a pair of integers and returns their sum (modulo 2^{64}) and $I(\texttt{~})$ is a function that takes a single 64-bit integer and returns its bitwise negation. I takes a pair of an update operator and an integer size and returns a function that takes two integers of this size and returns a third integer of this size. Note that the actual updating is not done by this function. For example, $I(\texttt{<<=},8)$ is a function that takes two 8-bit integers and returns the first rotated left by the second modulo 8. So $I(\texttt{<<=},8)(129,18) = I(\texttt{<<=},8)(129,2) = 6$. I is defined outside the semantic rules. Recall that comparison operators return 0 when the relation is false and $2^{64}-1$ when the relation is true.

The rule for variables and constants says that the size and location of a scalar variable or constant is found in the environment. The rule for array elements states that the location of the variable is bound in the store to a pair of vector size and vector elements, that the index expression must evaluate to a value less than the vector size, and that the location of the array element is found in the vector of elements. The type system guarantees that the location is not `null` and that it is bound to a pair, but it does not ensure that the index is within bounds, so this is checked at runtime. If the index it out of bounds, the effect is undefined.

The two first rules for expressions handle constants. The first handles simple number constants, which evaluate to themselves, and the second handles named constants that are bound to pairs of values and `null` locations. The rule for L-values finds the location of the L-value and gets its contents from the store, and then extends the value to 64 bits. For this, we use a postfix operator $\uparrow_z$ that extends a z-bit value to 64 bits. The rules for unary and binary operators evaluate the operand(s) and then applies the semantic operator to the value(s) of the operand(s). Finally, the rule for `size` finds the size of the array in the store. The type system ensures that the location is not `null` and that it is bound to a pair.

4.2 Statements

To handle `uncall` in the semantics for statements, we need to "run" statements backwards. To this end, we use the function I in Fig. 6 to invert statements:

In a type-correct program, the effect of first executing s and then $I(s)$ is, if s terminates without error, a null effect: The store is in the same state as before s was executed. Proving this is tedious, but relatively uncomplicated. The main complications are declarations and that some statements are only reversible if the aliasing constraint in th etype system hold. We do, however, not at the time have a complete proof written down.

Statements transform stores into stores, while keeping the environment unchanged. Sequents for running statements are of the form $\Delta, \eta \models_S s : \sigma_0 \rightleftharpoons \sigma_1$ and state that, given a procedure environment Δ and a variable environment η, a statement s reversibly transforms a store σ_0 to a store σ_1.

The rules for statements are shown in Fig. 7. The rule for the empty statement states that it does not change the store. The rule for updates finds the value v of the L-value and the value w of the expression. It then truncates w to s bits (using the $\downarrow_s$ operator), performs the operation (restricted to s bits) between the two values, and stores the result in the location of the L-value.

The rule for swap finds the values of the two L-values in the store and updates the store with these swapped. There are two rules for conditional swap: The first rule states that if the condition evaluates to 0 (false), there is no change in the store. The other rule states that if the condition evaluates to a non-zero (true) value, the effect on the store is like an unconditional swap. Note that this does not imply that the condition is evaluated twice if it is non-zero, nor that the timing differs. It is up to the implementation to ensure invariant timing.

The rule for loops first evaluate the loop bounds, allocates a new location in the store, and stores the first bound at the location, applies helper rules $\models_F$ using an environment where the loop counter is bound to the location, and then disposes of the location in the resulting store. There are two helper rules: One for when the loop counter is equal to the second bound, and one where it does not. Both use the location and the value of the second bound.

The rule for `call` finds the sized locations of the arguments, looks the procedure up in the procedure environment to get the list of parameter names and the body of the procedure. It then creates a new environment that binds the parameter names to the argument locations and executes the body in this environment. This implements call-by-reference parameter passing. The rule for `uncall` is similar, but it is the inverse of the body that is executed. The type system guarantees that the sizes of the given parameters are the same as the sizes of the declared parameters.

The rule for blocks uses the declarations to extend the environment and store, executes the body, and uses the declarations to restrict the store.

4.3 Declarations

The rules for declarations is shown in Fig. 8. There are two kinds of sequents for declarations: $\eta_0, \sigma_0 \models_D d \rightsquigarrow \eta_1, \sigma_1$ says that the declaration d extends η_0 and σ_0 to η_1 and σ_1. Conversely, $\eta_0, \sigma_0 \models_D^{inv} d \rightsquigarrow \eta_1, \sigma_1$ says that "undoing" the declaration d restricts η_0 to η_1 and σ_0 to σ_1.

$$\frac{}{\Delta, \eta \models_S \texttt{;} : \sigma \rightleftharpoons \sigma}\text{(Empty)}$$

$$\frac{\sigma, \eta \models_L l \mathbin{@} (z, \lambda) \quad \sigma(\lambda) = v_1 \quad \sigma, \eta \models_E e \to v_2}{\Delta, \eta \models_S l \oplus\!= e\texttt{;} : \sigma \rightleftharpoons \sigma[\lambda := I(\oplus\!=, z)(v_1, v_2 \downarrow_z)]}\text{(Update)}$$

$$\frac{\sigma, \eta \models_L l_1 \mathbin{@} (z, \lambda_1) \quad \sigma, \eta \models_L l_2 \mathbin{@} (z, \lambda_2) \quad \sigma(\lambda_1) = v_1 \quad \sigma(\lambda_1) = v_1}{\Delta, \eta \models_S l_1 \texttt{ <-> } l_2\texttt{;} : \sigma \rightleftharpoons \sigma[\lambda_1 := v_2,\ \lambda_2 := v_1]}\text{(Swap)}$$

$$\frac{\sigma, \eta \models_E e \to v \quad v = 0}{\Delta, \eta \models_S \texttt{if (}e\texttt{) } l_1 \texttt{ <-> } l_2\texttt{;} : \sigma \rightleftharpoons \sigma}\text{(CondSwap1)}$$

$$\frac{\begin{array}{c}\sigma, \eta \models_E e \to v \quad v \neq 0 \\ \sigma, \eta \models_L l_1 \mathbin{@} (z, \lambda_1) \quad \sigma, \eta \models_L l_2 \mathbin{@} (z, \lambda_2) \quad \sigma(\lambda_1) = v_1 \quad \sigma(\lambda_2) = v_2\end{array}}{\Delta, \eta \models_S \texttt{if (}e\texttt{) } l_1 \texttt{ <-> } l_2\texttt{;} : \sigma \rightleftharpoons \sigma[\lambda_1 := v_2,\ \lambda_2 := v_1]}\text{(CondSwap2)}$$

$$\frac{\begin{array}{c}\sigma, \eta \models_E e_1 \to v_1 \quad \sigma, \eta \models_E e_2 \to v_2 \\ (\sigma_1, \lambda) = \mathit{newlocation}_{64}(\sigma) \quad \sigma_2 = \sigma_1[\lambda := v_1] \\ \Delta, \eta[x \mapsto (64, \lambda)], \lambda, v_2 \models_F s : \sigma_2 \rightleftharpoons \sigma_3 \quad \sigma_4 = \mathit{disposelocation}_{64}(\sigma_3, \lambda)\end{array}}{\Delta, \eta \models_S \texttt{for (}x\texttt{=}e_1\texttt{; }e_2\texttt{) } s : \sigma \rightleftharpoons \sigma_4}\text{(ForLoop)}$$

$$\frac{\sigma(\lambda) = v}{\Delta, \eta, \lambda, v \models_F s : \sigma \rightleftharpoons \sigma}\text{(Loop1)}$$

$$\frac{\sigma(\lambda) \neq v \quad \Delta, \eta \models_S s : \sigma \rightleftharpoons \sigma_1 \quad \eta, \lambda, v \models_F s : \sigma_1 \rightleftharpoons \sigma_2}{\Delta, \eta, \lambda, v \models_F s : \sigma \rightleftharpoons \sigma_2}\text{(Loop2)}$$

$$\frac{\begin{array}{c}\forall i \in [1, n] : \sigma, \eta \models_L l_i @ (z_i, \lambda_i) \quad \Delta f = ([(x_1, z_1), \ldots, (x_n, z_n)], s) \\ \Delta, [x_1 \mapsto (z_1, \lambda_1), \ldots, x_n \mapsto (z_n, \lambda_n)] \models_S s : \sigma \rightleftharpoons \sigma_1\end{array}}{\Delta, \eta \models_S \texttt{call } f\texttt{(}l_1, \ldots, l_n\texttt{);} : \sigma \rightleftharpoons \sigma_1}\text{(Call)}$$

$$\frac{\begin{array}{c}\forall i \in [1, n] : \sigma, \eta \models_L l_i @ (z_i, \lambda_i) \quad \Delta f = ([(x_1, z_1), \ldots, (x_n, z_n)], s) \\ \Delta, [x_1 \mapsto (z_1, \lambda_1), \ldots, x_n \mapsto (z_n, \lambda_n)] \models_S I(s) : \sigma \rightleftharpoons \sigma_1\end{array}}{\Delta, \eta \models_S \texttt{call } f\texttt{(}l_1, \ldots, l_n\texttt{);} : \sigma \rightleftharpoons \sigma_1}\text{(Uncall)}$$

$$\frac{\begin{array}{c}\eta, \sigma \models_D d \rightsquigarrow \eta_0, \sigma_0 \\ \forall i \in [1, n] : \Delta, \eta_0 \models_S s_i : \sigma_{i-1} \rightleftharpoons \sigma_i \\ \eta_0, \sigma_n \models_D^{inv} d \rightsquigarrow \eta, \sigma_{n+1}\end{array}}{\Delta, \eta \models_S \{d\ s_1 \ldots s_n\} : \sigma_0 \rightsquigarrow \sigma_{n+1}}\text{(Block)}$$

Fig. 7. Semantic rules for statements

The first two rules say that the empty declaration has no effect. The next two rules state that a constant declaration extends the environment but leaves the store unchanged. Recall that constants are stored in the environment by using a null location. The rules for variable and array declarations do not distinguish secret and public values. In the forwards direction, a new location (bound to zero) is created for the variable and the variable is bound to the location. In the backwards direction, $\mathit{disposelocation}_z$ verifies that the location is bound to zero before it is removed from the store. In the forwards direction, a new

$$\frac{}{\eta, \sigma \models_D \rightsquigarrow \eta, \sigma}(\text{EmptyDecl}) \qquad \frac{}{\eta, \sigma \models_D^{inv} \rightsquigarrow \eta, \sigma}(\text{EmptyDeclInv})$$

$$\frac{\eta[x \mapsto (n, \mathsf{null})], \sigma \models_D d \rightsquigarrow \eta_1, \sigma_1}{\eta, \sigma \models_D \texttt{const}\ \ x\texttt{=}n\texttt{;}d \rightsquigarrow \eta_1, \sigma_1}(\text{ConstDecl})$$

$$\frac{\eta, \sigma \models_D^{inv} d \rightsquigarrow \eta_1, \sigma_1 \quad \eta_2[x \mapsto (n, \mathsf{null})] = \eta_1}{\eta, \sigma \models_D^{inv} \texttt{const}\ \ x\texttt{=}n\texttt{;}d \rightsquigarrow \eta_2, \sigma_1}(\text{ConstDeclInv})$$

$$\frac{(\sigma_1, \lambda) = newlocation_z(\sigma) \quad \eta[x \mapsto (z, \lambda)], \sigma_1 \models_D d \rightsquigarrow \eta_2, \sigma_2}{\eta, \sigma \models_D t\ \ \texttt{u}z\ x\texttt{;}d \rightsquigarrow \eta_2, \sigma_2}(\text{VarDecl})$$

$$\frac{\begin{array}{c}\eta, \sigma \models_D^{inv} d \rightsquigarrow \eta_1, \sigma_1 \\ \eta_2[x \mapsto (z, \lambda)] = \eta_1 \quad \sigma_2 = disposelocation_z(\sigma_1, \lambda)\end{array}}{\eta, \sigma \models_D^{inv} t\ \ \texttt{u}z\ x\texttt{;}d \rightsquigarrow \eta_2, \sigma_2}(\text{VarDeclInv})$$

$$\frac{\begin{array}{c}\sigma, \eta \models_E e \rightarrow n \quad (\sigma_1, \lambda) = newarray_z(\sigma, n) \\ \eta[x \mapsto (z, \lambda)], \sigma_1 \models_D d \rightsquigarrow \eta_2, \sigma_2\end{array}}{\eta, \sigma \models_D t\ \ \texttt{u}z\ x\texttt{[}e\texttt{];}d \rightsquigarrow \eta_2, \sigma_2}(\text{ArrayDecl})$$

$$\frac{\begin{array}{c}\eta, \sigma \models_D^{inv} d \rightsquigarrow \eta_1, \sigma_1 \\ \sigma_1, \eta \models_E e \rightarrow n \quad \eta_2[x \mapsto (z, \lambda)] = \eta_1 \quad \sigma_1(\lambda) = (n, ve) \\ \sigma_2 = disposearray_z(\sigma_1, z, \lambda)\end{array}}{\eta, \sigma \models_D^{inv} t\ \ \texttt{u}z\ x\texttt{[}e\texttt{];d} \rightsquigarrow \eta_2, \sigma_2}(\text{ArrayDeclInv})$$

Fig. 8. Semantic rules for declarations

zeroed array is created in the store and the variable is bound to its location in the environment. In the backwards direction is it verified that the expression evaluates to the array size, and $disposearray_z$ checks that the elements of the array are all bound to 0 in the store and removes the array from the store. Note that the rules for undeclaring things treat the declarations in reverse order.

4.4 Procedures and Programs

The rules for procedures and programs are shown in Fig. 9. There is no `main` function and no input/output in Hermes, so it is assumed that procedures are called from outside Hermes. Therefore, the semantics of a program is just creating a procedure environment Δ. The external program can call (or uncall) a procedure in this environment by providing a store and a list of locations for the procedure parameters. The rule for procedures creates a procedure environment for a single procedure. This binds the procedure name to a list of (name, integer size) pairs and the body of the procedure. The environments are combined using $\uplus$ in the rule for programs. Additional rules describe external calls to Hermes. These are very like the rules for calls in statements, except that the locations are given directly instead of being derived from a list of L-values.

$$\frac{\forall i \in [1,n] : \models_P p_i \gg \Delta_i}{\models p_1 \ldots p_n \gg \Delta_1 \uplus \cdots \uplus \Delta_n}\text{(Program)} \qquad \frac{\models_A a \hookrightarrow xs}{\models_P f(a)\ s \gg [f \mapsto (xs, s)]}\text{(Procedure)}$$

$$\frac{\models_A a_1 \hookrightarrow xs_1 \quad \models_A a_2 \hookrightarrow xs_2}{\models_A a_1\ ,\ a_2 \hookrightarrow xs_1 \uplus xs_2}\text{(ArgList)}$$

$$\frac{}{\models_A t\ \texttt{uz}\ x \hookrightarrow [(x,z)]}\text{(Scalar)} \qquad \frac{}{\models_A t\ \texttt{uz}\ x\texttt{[]} \hookrightarrow [(x,z)]}\text{(Array)}$$

$$\frac{\Delta f = ([(x_1,z_1),\ldots,(x_n,z_n)], s) \quad \Delta, [x_1 \mapsto (z_1,\lambda_1),\ldots,x_n \mapsto (z_n,\lambda_n)] \models_S s : \sigma \rightleftharpoons \sigma_1}{\Delta, \sigma, [\lambda_1,\ldots,\lambda_n] \models_X \texttt{call}\ f \Rightarrow \sigma_1}\text{(Xcall)}$$

$$\frac{\Delta f = ([(x_1,z_1),\ldots,(x_n,z_n)], s) \quad \Delta, [x_1 \mapsto (z_1,\lambda_1),\ldots,x_n \mapsto (z_n,\lambda_n)] \models_S R(s) : \sigma \rightleftharpoons \sigma_1}{\Delta, \sigma, [\lambda_1,\ldots,\lambda_n] \models_X \texttt{uncall}\ f \Rightarrow \sigma_1}\text{(Xuncall)}$$

Fig. 9. Semantic rules for procedures and programs

5 Code Examples

In the examples, we use some syntactic sugar that the Hermes compiler expands into the core syntax during parsing. The statements *Lval*`++;`, *Lval*`--;`, and `if (`*Exp*`)` *Lval* **update** *Exp*`;` are expanded to *Lval* `+= 1;`, *Lval* `-= 1;`, and *Lval* **update** `(`*Exp* `!= 0) & (`*Exp*`);`, respectively. The latter works because 0 is a neutral element for all the update operators used in Hermes. A declaration that specifies a number of variables and arrays of the same type is expanded to a sequence of individual declarations, and if `secret` or `public` is omitted from a declaration, `secret` is assumed. For example, the declarations `public u32 x, a[n]; u64 z;` is just a shorter way to write the equivalent `public u32 x; public u32 a[n]; secret u64 z;`. Operator precedences can be overridden by parentheses.

Figure 10 (top) shows Hermes code for the TEA encryption algorithm [14], a simple cypher used mainly for teaching. Only the encryption function is shown – decryption is done by `uncall`ing the encryption function. The sizes of `v` and `k` are 2 and 4, respectively. Compare to the equivalent program in C [17] at the bottom of Fig. 10. Apart from using updates and swaps, the main difference is that the C version requires an explicit decryption function, which is not needed in Hermes. Also, the local variables are in Hermes cleared to 0 by "uncomputation", where the C version leaves these uncleared.

Figure 11 shows Hermes and C code for the central part of RC5 [9], another simple algorithm. The Hermes program shows `size s` being used as a loop bound, which makes the procedure independent of the size of the expanded key. Since C does not have a rotate operator, the C version [15] uses a macro for this. And since C does not have a swap operator, the central loop is unrolled so one iteration in the C version correspond to two iterations in the Hermes version. Again, C needs an explicit decryption function (not shown), which is not required in Hermes. Key expansion in RC5 (not shown) is not reversible, so to

```
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, now clear local variables */
    k0 -= k[0]; k1 -= k[1]; k2 -= k[2]; k3 -= k[3]; sum -= 0xC6EF3720;
    v[0] <-> v0; v[1] <-> v1;              /* return coded values */
}
```

```
void encrypt (uint32_t v[2], uint32_t k[4]) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;   /* set up */
    uint32_t delta=0x9E3779B9;             /* key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];  /* cache key */
    for (i=0; i<32; i++) {                  /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                       /* end cycle */
    v[0]=v0; v[1]=v1;
}

void decrypt (uint32_t v[2], uint32_t k[4]) {
    uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i; /* sum=32*delta */
    uint32_t delta=0x9E3779B9;             /* key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];  /* cache key */
    for (i=0; i<32; i++) {                  /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                       /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Fig. 10. TEA in Hermes (top) and C (bottom)

implement this in Hermes requires storing additional values i "garbage" array. Th garbage array is reset to zeroes when the expanded key (after calling the central procedure) is uncomputed by `uncall`ing the key expansion procedure.

Figure 12 shows Hermes code for speck128 [1,18] (a cypher used by NSA). Again, only encoding is shown. The main thing to note is that the `R` procedure are found in two copies, one (`Rs`) where the `k` parameter is secret, and one (`Rp`) where it is public. This is because two of the calls pass a public loop counter to `k`, while the other two calls pass part of a secret key to `k`. An extension to the type system that avoids this codeduplication is being investigated. Some

```
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++;
  }
ct[0] <-> A; ct[1] <-> B;
}
```

```
#define ROL(x,r) ((x<<r)|(x>>(64-r)))
void RC5ENCRYPT(WORD *pt, WORD *ct)
{
  WORD i, A=pt[0]+S[0], B=pt[1]+S[1];
  for(i = 1; i <= 12; i++)
  {
    A = ROL(A ^ B, B) + S[2*i];
    B = ROL(B ^ A, A) + S[2*i + 1];
  }
  ct[0] = A; ct[1] = B;
}
```

Fig. 11. RC5 core in Hermes (left) and C (right)

```
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; 32) {
    call Rp(a, b, i); i++;
    call Rs(x, y, b);
  }
  for (i=32; 0) {  /* restore a and b */
    i--; uncall Rp(a, b, i);
  }
  y <-> ct[0]; x <-> ct[1]; b -= K[0]; a -= K[1];
}

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; }
```

Fig. 12. Speck128 in Hermes

uncomputation is needed to restore `a` and `b` to 0. This is not found in the standard C implementation, where these are left uncleared.

We have implemented several other encryption algorithms in Hermes, including Red Pike [16] (a cypher used by GCHQ) and Blowfish [10] (designed as a replacement for DES). With the exception of key expansion, this was relatively straight forward.

6 Conclusion and Future Work

We have presented a language Hermes for writing light-weight encryption functions. Hermes ensures reversibility, so decryption can be done by executing

encryption procedures backwards, and can (given a suitable implementation) protect against certain forms of side-channel attacks, such as timing based attacks and leaks to memory. Hermes has a formal semantics for both the type system and runtime behavior. These semantics can be used to prove both that secret information does not leak into publica variables and that type-correct programs are, indeed, reversible, but we do not have complete proofs for this at the moment, mainly because we expect Hermes to evolve over time, so we have postponed proofs until Hermes settles to a more stable form. The semantic rules do not specify what happens if a condition in a rule fails, for example when an array bound is exceeded. For the type rules, the obvious behaviour is an error message. For the run-time semantics, it is less clear. Run-time error messages can be helpful in locating errors, but they can potentially leak information about secret values. So it might be better to continue execution with some default behaviour.

We have in Standard ML made a reference interpreter for Hermes which closely follows the semantic rules. The interpreter does not guarantee time-invariant operations, and it reports errors when run-time errors are detected. We also have an implementation of Hermes in WebAssembly [2]. We are working on extending this to target CT-Wasm [13], a variant of WebAssembly that has a public/secret type system similar to the one used here. Targeting CT-Wasm should preserve the safety features of Hermes. Note that the aliasing restrictions in Hermes make call-by-reference indistinguishable from call-by-value-return, so this can be used as an optimisation when WebAssembly, as planned, supports multiple return values.

We are currently working on implementing the Advanced Encryption Standared (AES) in Hermes. An issue with AES is that it uses secret information as array indexes, which the current Hermes does not allow, so to implement it may require a relaxation of this restriction, for example by ensuring the array is fully cached, so access time is independent of the index. We are also considering other extensions to Hermes, including sized boolean types (with values 0 and 2^z-1) and read-only parameters to procedures. The latter will avoid the need of duplicating the R procedure in Fig. 12. We are also considering additional control structures, but will only add them by need. A more precise alias analysis could relax some of the restrictions on parameter passing, but we have not found any examples where this matters. At the moment, index checks and checks that variables and arrays are zeroed before being disposed are done at run time. Static verification of these would be beneficial, for efficiency and safety both.

Some side-channel attacks (such as Spectre [4]) target speculative execution. By partially evaluating [3,6] Hermes programs with all public values (typically key and block lengths) considered static will leave a straight-line unconditional sequence of operations only involving secret values and constants, thus avoiding speculative execution. This has the added benefit that it is easier to eliminate index checks and checks for variables being zero at the end of blocks.

Public-key cyphers are not trivially reversible – that would defeat the purpose – so implementing these in Hermes it not obvious. A possibility is to let

the encryption function return not only the cypher text, but also additional "garbage" information that must be discarded before transmitting the cypher text. Similarly, decryption also produces garbage in addition to the original text. As such, the reversibility of Hermes is not exploited, but is rather a hindrance. The safety features still apply, though.

We thank our colleagues Ken Friis Larsen and Michael Kirkedal for co-supervising some student projects about Hermes and for fruitful discussions, and we thank the students who worked on these projects.

References

1. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404 (2013). https://eprint.iacr.org/2013/404
2. Haas, A., et al.: Bringing the web up to speed with Webassembly. SIGPLAN Not. **52**(6), 185–200 (2017)
3. Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.): DIKU 1998. LNCS, vol. 1706. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47018-2
4. Kocher, P., et al.: Exploiting speculative execution. Spectre attacks (2018). meltdownattack.com
5. Lutz, C.: Janus: a time-reversible language. A letter to Landauer (1986). http://www.tetsuo.jp/ref/janus.pdf
6. Mogensen, T.Æ: Partial evaluation of the reversible language Janus. In: Khoo, S.-C., Siek, J.G. (eds.) PEPM 2011, pp. 23–32. ACM (2011)
7. Mogensen, T.Æ.: Hermes: a reversible language for writing encryption algorithms (work in progress). In: Bjørner, N., Virbitskaite, I., Voronkov, A. (eds.) PSI 2019. LNCS, vol. 11964, pp. 243–251. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37487-7_21
8. Palsberg, J., Ørbæk, P.: Trust in the λ-calculus. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 314–329. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60360-3_47
9. Rivest, R.L.: The RC5 encryption algorithm. Dr. Dobb's J. **20**(1), 146–148 (1995)
10. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 191–204. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58108-1_24
11. Smith, G.: Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection, pp. 291–307. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-44599-1_13
12. Táborský, D., Larsen, K.F., Thomsen, M.K.: Encryption and reversible computations. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 331–338. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_23
13. Watt, C., Renner, J., Popescu, N., Cauligi, S., Stefan, D.: CT-Wasm: type-driven secure cryptography for the web ecosystem. CoRR, abs/1808.01348 (2018)
14. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60590-8_29
15. Wikipedia: RC5. https://en.wikipedia.org/wiki/RC5. Accessed Feb 2019
16. Wikipedia: Red pike (cipher). https://en.wikipedia.org/wiki/Red_Pike_(cipher). Accessed Feb 2019

17. Wikipedia: Tiny encryption algorithm. https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm. Accessed Jan 2019
18. Wikipedia: Speck (cipher). https://en.wikipedia.org/wiki/Speck_(cipher). Accessed Feb 2019
19. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of the 5th Conference on Computing frontiers, CF 2008, pp. 43–54. ACM, New York (2008)