Hermes: A Reversible Language for Lightweight Encryption

Torben Ægidius Mogensen

DIKU, University of Copenhagen
Universitetsparken 5, DK-2100 Copenhagen O, Denmark

Abstract

Hermes is a domain-specific language for writing lightweight encryption algorithms: It is reversible, so it is not necessary to write separate encryption and decryption procedures. Hermes uses a type system that 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 lightweight encryption algorithms in Hermes and C.

Keywords: `lightweight encryption, side-channel attacks, reversible programming languages, domain-specific languages`

1. Introduction

Recent work [1] has investigated using the reversible language Janus [2, 3] 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 encryption functions in a reversible programming language. Additionally, reversible languages requires that all intermediate variables are cleared to 0 (or some other known constant) before they are discarded, which ensures that no information that could potentially be used for side-channel attacks is left in memory. 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 or 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. Nor does Janus have a clear separation between secret and public values.

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. Earlier versions of the Hermes language were presented at PSI'19 [4] and RC 2020 [5]. Experiments using the 2019 version language indicated a need for a type system that separates secret and public data, which was added to the 2020 version. In the 2019 version, the type system only distinguishes constants, loop counters, and everything else, with constants and loop variables being considered non-secret and all else being secret. This was, 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.

The 2020 version of Hermes 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 [6], trust analysis [7], and information flow analysis [8] but also imposes restrictions to ensure reversibility and (under reasonable assumptions) avoid timing-based side-channel attacks.

Working with the 2020 version of Hermes, including using Hermes to implement the Advanced Encryption Standard (AES) [9], has suggested further changes to the language. In particular (as noted above), to ensure that operations on secret values use time independent of the actual values, Hermes restricts array indexes to be public data (since reading from cached and uncached addresses take different time), but AES does, in fact, use secret values to index arrays (S-boxes). This is a known weakness of AES and other encryption methods that use S-boxes [10]. To allow AES and other encryptions using S-boxes to be implemented in Hermes, we have added a keyword `unsafe` to array look-ups, which allows secret values as indexes. The `unsafe` keyword should be used with caution (as the name indicates), as even caching the array before use does not guarantee constant timing. Nevertheless, implementations of Hermes should do their best to make `unsafe` look-ups as close to constant time as possible. See more details in Section 5.4.

Other extensions include an reversible if-then-else construct (with public control). While such can be emulated using the for-loop construct in Hermes, this is not very efficient.

2. Hermes Syntax

The core syntax of Hermes is shown in Figure 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.

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

Figure 1: Core syntax of Hermes

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. We have found no need for signed integers in any of the ciphers we have investigated, so these are omitted.

unOp denotes an unary operator on numbers. This can be bit-wise 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 bit-wise logical operators.

update denotes an update operator. This can be one of `+=`, `-=`, `^=`, `<<=`, and `>>=`. The first three operators have the same meaning as in C. `<<=` and `>>=` are left and right rotates. 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. 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 public 64-bit number. So we have:

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

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 with z bits. We define a partial order $\sqsubseteq$ as the reflexive extension of $\texttt{public} \sqsubset \texttt{secret}$ and a least upper bound operator $\sqcup$ induced by this partial order. We use this to make the result of an operation 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.

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

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

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

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

For L-values, the rule for variables states 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. Unsafe array look-ups can use both public and secret indexes, but the elements of the array must be secret. This is because the result of look-ups and updates may depend on secret values.

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 two rules states that the size of an array is a public value.

3.2. Statements and local declarations

A sequent 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 Figure 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 that the root variable of the L-val can not occur in the expression nor in an

$$\frac{}{\rho \vdash_L x : \rho(x)}\text{(Variable)}$$

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

$$\frac{\rho(x) = \texttt{secret}^z\,\texttt{[]} \quad \rho \vdash_E e : t}{\rho \vdash_L \texttt{unsafe}\ x[e] : \texttt{secret}^z}\text{(UnsafeArrayAccess)}$$

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

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

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

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

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

Figure 2: Type rules for L-values and expressions

index expression of the L-val. For example, the updates `x -= x;`, `a[i] <<= a[j];`, and `a[a[i]] += 1;` are disallowed. 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. A swap of two elements of the same array `a[i] <-> a[j];` is, however, allowed.

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. Note that conditional swap can use a secret condition (as long as both L-values are secret), which requires that it must be implemented in a time-invariant fashion, e.g., that `if (c) x<->y` is implemented by code equivalent to the C code

```
tmp = c & (x ^ y);
x ^= tmp;
y ^= tmp;
tmp = 0;
```

the last assignment is to clear to zero the possibly secret value of tmp.

The condition in the more general if-then-else statement must be public to ensure that the timing does not depend on secret data. To make the if-then-else statement reversible, the condition must evaluate to the same in both the forwards and the backwards direction, which means that no variable or array used in the condition can be updated in the branches of the if-then-else. An alternative is to use a Janus-style conditional with different entry and exit conditions, but that requires and extra run-time test, and it can easily be emulated by the for loop without significant overhead, so we have chosen the simpler (and more efficient) form.

We define $U(s)$ to be the set of variables and arrays that can be updated in a statement s and $D(d)$ to be the set of variables, constants, and arrays declared by a declaration by

$$\begin{array}{lcl}
U(\texttt{;}) & = & \emptyset \\
U(l \oplus\!= e\texttt{;}) & = & \{R(l)\} \\
U(l_1 \texttt{ <-> } l_2\texttt{;}) & = & \{R(l_1),\, R(l_2)\} \\
U(\texttt{if (}c\texttt{) } l_1 \texttt{ <-> } l_2\texttt{;}) & = & \{R(l_1),\, R(l_2)\} \\
U(\texttt{if (}c\texttt{) } s_1 \texttt{ else } s_2) & = & U(s_1) \cup U(s_2) \\
U(\texttt{for (}x\texttt{=}e_1\texttt{; } e_2\texttt{) } s) & = & U(s) \setminus \{x\} \\
U(\texttt{call } f(l_1,\, \ldots,\, l_n)\texttt{;}) & = & \{R(l_1),\, \ldots,\, R(l_n)\} \\
U(\texttt{uncall } f(l_1,\, \ldots,\, l_n)\texttt{;}) & = & \{R(l_1),\, \ldots,\, R(l_n)\} \\
U(\{d\ s_1\ \ldots\ s_n\}) & = & (U(s_1) \cup \cdots \cup U(s_n)) \setminus D(d)
\end{array}$$

$$\begin{array}{lcl}
D() & = & \emptyset \\
D(t\ \texttt{u}z\ x\texttt{;}\ d) & = & \{x\} \cup D(d) \\
D(\texttt{constant } x\texttt{=}n\texttt{;}\ d) & = & \{x\} \cup D(d) \\
D(t\ \texttt{u}z\ x\texttt{[}e\texttt{];}\ d) & = & \{x\} \cup D(d)
\end{array}$$

The rule for for loops state that the loop bounds must be public, that the variables used in the bounds expressions can not be modified in the loop body, 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 anywhere in another argument, nor in the index expression of the same argument. These are similar to the restrictions on updates since the called procedure can use any parameter to update any other.

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.

Figure 4 show the rules for how declarations extend environments.

Sequents for declarations are of the form $\rho \vdash_D d \rightsquigarrow \rho_1$, and state that the

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

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

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

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

$$\frac{\rho \vdash_E e : \texttt{public} \quad \Gamma, \rho \vdash_S s_1 \quad \Gamma, \rho \vdash_S s_2 \quad V(e) \cap (U(s_1) \cup U(s_2)) = \emptyset}{\Gamma, \rho \vdash_S \texttt{if (}e\texttt{)}\, s_1 \,\texttt{else}\, s_2}\text{(IfElse)}$$

$$\frac{\begin{array}{c}\rho \vdash_E e_1 : \texttt{public} \quad \rho \vdash_E e_2 : \texttt{public} \\ (V(e_1) \cup V(e_2)) \cap U(s) = \emptyset \quad \Gamma, \rho[x \mapsto \texttt{public}^{64}] \vdash_S s\end{array}}{\Gamma, \rho \vdash_S \texttt{for (}x \texttt{=} e_1 \texttt{;}\, e_2\texttt{)}\, s}\text{(ForLoop)}$$

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

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

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

Figure 3: Type rules for statements

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

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

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

Figure 4: Type rules for declarations

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 Figure 5. A sequent of the form $\vdash pgm$ states that pgm is a valid program. $\vdash_P p \rightsquigarrow \Gamma$ states 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 $Procedure1$ rule builds a procedure environment for a single procedure, and $Procedure2$ 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.

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

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

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

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

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

Figure 5: 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. The run-time semantics also does not specify timing. It is up to implementations to ensure invariant timing of operations on secret values – except unsafe array look-ups, and even those must be made as close to invariant as possible.

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. In this way, the run-time semantics are more akin to an interpreter than to a compiler.

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. This means that removing a location from the top of the stack and then immediately allocating a same-sized location will yields the same location that was disposed. This is true also of a sequence of locations: Disposing a sequence of locations and then allocating same-sized locations in the opposite order will yield the same sequence of locations.

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 a 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 $(\sigma_1, \lambda) = \mathit{newlocation}_z(\sigma)$, then $\sigma = \mathit{disposelocation}_z(\sigma_1 \lambda)$, and if $\sigma = \mathit{disposelocation}_z(\sigma_1 \lambda)$ is defined, then $(\sigma_1, \lambda) = \mathit{newlocation}_z(\sigma)$.

We also use a family of functions $\mathit{newarray}_z$ 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. $\mathit{newarray}_z$ also have duals, $\mathit{disposearray}_z$, 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 $(\sigma_1, \lambda) = \mathit{newarray}_z(\sigma, vs)$, then $\sigma = \mathit{disposearray}_z(\sigma_1, vs, \lambda)$, and if $\sigma = \mathit{disposearray}_z(\sigma_1, vs, \lambda)$ is defined, then $(\sigma_1, \lambda) = \mathit{newarray}_z(\sigma, vs)$, then $\sigma = \mathit{disposearray}_z(\sigma_1, vs, \lambda)$.

4.1. L-values and expressions

Figure 6 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 $\sigma, \eta \models_L l \,@\, (z, \lambda)$ 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.

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

We use a function O that binds operator symbols to the functions they represent. So $O(\texttt{+})$ is a function that takes a pair of integers and returns their sum (modulo 2^{64}) and $O(\texttt{~})$ is a function that takes a single 64-bit integer and returns its bit-wise negation. U 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, $U(\texttt{<<=}, 8)$ is a function that takes two 8-bit integers and returns the first rotated left by the second modulo 8. So $U(\texttt{<<=}, 8)(129, 18) =$

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

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

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

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

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

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

Figure 6: Semantic rules for L-values and expressions

$U(\texttt{<<=}, 8)(129, 2) = 6$. O and U are 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 run-time. If the index it out of bounds, the effect is undefined. Unsafe array look-ups have the same run-time semantics as normal look-ups.

The two first rules for expressions handle constants. The first handles simple

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

$$\begin{array}{rcl}
I(l_1\,\texttt{<->}\,l_2\texttt{;}) & = & l_1\,\texttt{<->}\,l_2\texttt{;} \\
I(\texttt{if}\ \texttt{(}c\texttt{)}\ l_1\,\texttt{<->}\,l_2\texttt{;}) & = & \texttt{if}\ \texttt{(}c\texttt{)}\ l_1\,\texttt{<->}\,l_2\texttt{;} \\
I(\texttt{if}\,\texttt{(}c\texttt{)}\ s_1\ \texttt{else}\ s_2\texttt{;}) & = & \texttt{if}\,\texttt{(}c\texttt{)}\ I(s_1)\ \texttt{else}\ I(s_2)\texttt{;} \\
I(\texttt{for}\ \texttt{(}x\,\texttt{=}\,e_1\texttt{;}\,e_2\texttt{)}\ s) & = & \texttt{for}\ \texttt{(}x\,\texttt{=}\,e_2\texttt{;}\,e_1\texttt{)}\ I(s) \\
I(\texttt{call}\ f\texttt{(}as\texttt{);} & = & \texttt{uncall}\ f\texttt{(}as\texttt{);} \\
I(\texttt{uncall}\ f\texttt{(}as\texttt{);} & = & \texttt{call}\ f\texttt{(}as\texttt{);} \\
I(\{d\ s_1 \ldots s_n\}) & = & \{d\ I(s_n) \ldots I(s_1)\}
\end{array}$$

Figure 7: Inverting statements

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 post-fix 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 Figure 7 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. A semi-formal proof of this is shown in Appendix A. Note that $I(I(s)) = s$.

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 Figures 8 and 9. 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 rules for the if-then-else statement are straight-forward: If the condition evaluates to a non-zero value, the first statement is executed, otherwise the second statement is executed.

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 values of the bounds. Note that, to ensure reversibility, the loop counter can after the first iteration not be equal to its initial value.

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 en-

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

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

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

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

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

$$\frac{\sigma, \eta \models_E e \to v \quad v \neq 0 \quad \Delta, \eta \models_S s_1 : \sigma_1 \rightleftharpoons \sigma_2}{\Delta, \eta \models_S \texttt{if(}e\texttt{) } s_1 \texttt{ else } s_2\texttt{;} : \sigma_1 \rightleftharpoons \sigma_2}(If1)$$

$$\frac{\sigma, \eta \models_E e \to v \quad v = 0 \quad \Delta, \eta \models_S s_2 : \sigma_1 \rightleftharpoons \sigma_2}{\Delta, \eta \models_S \texttt{if(}e\texttt{) } s_1 \texttt{ else } s_2\texttt{;} : \sigma_1 \rightleftharpoons \sigma_2}(If2)$$

$$\frac{\begin{array}{c}\sigma, \eta \models_E e_1 \to v_1 \quad \sigma, \eta \models_E e_2 \to v_2 \\ (\sigma_1, \lambda) = newlocation_{64}(\sigma) \quad \sigma_2 = \sigma_1[\lambda := v_1] \\ \Delta, \eta[x \mapsto (64, \lambda)], \lambda, v_1, v_2 \models_F s : \sigma_2 \rightleftharpoons \sigma_3 \\ \sigma_4 = \sigma_3[\lambda := 0] \quad \sigma_5 = disposelocation_{64}(\sigma_4, \lambda)\end{array}}{\Delta, \eta \models_S \texttt{for (}x\texttt{=}e_1\texttt{; }e_2\texttt{) } s : \sigma \rightleftharpoons \sigma_5}\text{(ForLoop)}$$

$$\frac{\sigma(\lambda) = v_2}{\Delta, \eta, \lambda, v_1, v_2 \models_F s : \sigma \rightleftharpoons \sigma}\text{(Loop1)}$$

$$\frac{\begin{array}{c}\sigma(\lambda) \neq v_2 \quad \Delta, \eta \models_S s : \sigma \rightleftharpoons \sigma_1 \quad \sigma_1(\lambda) \neq v_1 \\ \eta, \lambda, v_1, v_2 \models_F s : \sigma_1 \rightleftharpoons \sigma_2\end{array}}{\Delta, \eta, \lambda, v_1, v_2 \models_F s : \sigma \rightleftharpoons \sigma_2}\text{(Loop2)}$$

Figure 8: Semantic rules for statements (part 1)

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

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

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

Figure 9: Semantic rules for statements (part 2)

vironment. 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 the store.

4.3. Declarations

The rules for declarations is shown in Figure 10. 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. Dually, $\eta_0, \sigma_0 \models_D^{inv} d \rightsquigarrow \eta_1, \sigma_1$ says that "undoing" the declaration d restricts η_0 to η_1 and σ_0 to σ_1.

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 (that is done by the type system). In the forwards direction, a new location (bound to zero) is created for the variable and the

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

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

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

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

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

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

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

Figure 10: Semantic rules for declarations

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 zeroed array is created in the store and the variable is bound to its location in the environment. In the backwards direction, $\mathit{disposearray}_z$ checks that the array size is as declared and that the elements of the array are all bound to 0 in the store, and then removes the array from the store. Note that the rules for undeclaring things treat the declarations in reverse order.

$$\frac{\forall i \in [1,n]: \quad \models_P p_i \gg \Delta_i}{\models p_1 \ldots p_n \gg \Delta_1 \uplus \cdots \uplus \Delta_n}\text{(Program)}$$

$$\frac{\models_A a \hookrightarrow xs}{\models_P f(a)\ s \gg [f \mapsto (xs, s)]}\text{(Procedure)}$$

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

$$\frac{}{\models_A t\ \ \mathtt{u}z\ x \hookrightarrow [(x,z)]}\text{(Scalar)}$$

$$\frac{}{\models_A t\ \ \mathtt{u}z\ x\mathtt{[]} \hookrightarrow [(x,z)]}\text{(Array)}$$

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

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

Figure 11: Semantic rules for procedures and programs

4.4. Procedures and programs

The rules for procedures and programs are shown in Figure 11. 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.

5. Code Examples

We have shown a number of code examples that compare C and Hermes in Appendix B, and we discuss these below.

The core language syntax in Figure 1 does not specify operator precedence or parentheses. We will use parentheses and the following operator precedences in the code examples discussed below.

`~`	prefix
`*  /  %`	left associative
`<<  >>`	left associative
`+  -`	left associative
`==  !=  <  >  <=  >=`	left associative
`&`	left associative
`|  ^`	left associative

Note that the precedences of comparison operators differ from those in C.

Additionally, the Hermes parser accepts a few constructs that during parsing are expanded into the core language:

The statements l`++;` and l`--;` are expanded to l `+= 1;` and l `-= 1;`, respectively.

`if` `(`e_1`)` $l \oplus\!=\ e_2$`;` is expanded to $l \oplus\!=$ `(`e_1 `!=0) & (`e_2`);`. This works because 0 is a neutral element for all the update operators used in Hermes and the result of comparisons is always either 0 or the neutral element for bit-wise conjunction (i.e., all bits set). Note that if the condition is already a comparison, the `!=0` part can be omitted. Additionally, `if (`e`)` s, where s is neither an update nor a swap, is expanded to `if (`e`)` s `else ;`.

A declaration that specifies a comma-separated list 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;`.

A common coding pattern in Hermes programs is $A\,B\,I(A)$, where $I(A)$ is the inverse of A, so for brevity and clarity we have added a shorthand for this: A `@` B. This is expanded into the full form in the parser. `@` is right-associative, so A `@` B `@` C is equivalent to A `@` $\{B$ `@` $C\}$ and expands to $A\,B\,C\,I(B)\,I(A)$.

Most of these syntactic extensions are used in the examples discussed below.

5.1. TEA

Figure B.13 (top) shows Hermes code for the TEA encryption algorithm [11], a simple cipher 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 [12] at the bottom of Figure B.13. 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, thereby potentially leaking information. The latter is easily fixed in the C program by explicitly zeroing the variables, but that requires that the C compiler does not remove assignments to dead variables.

5.2. RC5

Figure B.12 shows Hermes and C code for the central part of RC5 [13], another simple cipher. 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 [14] uses a macro for this (which most C compilers can optimise to a rotate). 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 implement this in Hermes requires storing additional values in a "garbage" array. The garbage array is reset to zeroes when the expanded key (after calling the central procedure) is uncomputed by `uncall`ing the key expansion procedure. If several blocks are encrypted using the same key (the typical situation), the key expansion and unexpansion need only be done once.

5.3. Speck128

Figure B.14 shows code in C (top) and Hermes (bottom) for speck128 [15, 16] (a cipher 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`. Some uncomputation is needed to restore `a` and `b` to 0. This is not found in the standard C implementation, where these are left uncleared.

5.4. AES and S-boxes

AES and several other encryption algorithms use S-Boxes. An S-box is a table of values that represents a permutation. AES uses a 256-byte S-boxes S that represents a constant permutation of 8-bit values. AES uses additional S-boxes to represent multiplication by constants in the field GF(2^8). These are used in a matrix multiplication in the field GF(2^8) by the matrix

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

This matrix multiplication is called *the mix-columns step.* Note that the 01s in the matrix do not require table look-ups, as they represent multiplication with the neutral element of multiplication.

The essence of AES is using parts of the key and message as indexes into such S-boxes. Since the S-boxes are permutations, this is reversible: If S is a 256-byte S-box, we can (assuming $Sinv$ is initialised to 0) compute the inverse S-box $Sinv$ by a simple loop:

```
for (i = 0; 256) {
  Sinv[S[i]] += i;
  i++;
}
```

Given this, replacing an 8-bit variable x by $S[x]$ can be done reversibly by the Hermes code

```
{ u8 x1;
  x1 += S[x];
  x -= Sinv[x1];
  x <-> x1;
}
```

Inverting the mix-columns step is done by multiplying by the inverse matrix

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

Since array look-ups are not guaranteed to be constant time, using a secret value (i.e., a part of the key or message) as index is unsafe, so unsafe array look-ups must be used for S-box look-ups. Unsafe look-ups imply that the array itself must be declared to have secret content, so both S and $Sinv$ must be declared as secret and the two code snippets above must be modified to use unsafe look-ups whenever secret data is used as index:

```
for (i = 0; 256) {
  unsafe Sinv[S[i]] += i;
  i++;
}

{ u8 x1;
  x1 += unsafe S[x];
  x -= unsafe Sinv[x1];
  x <-> x1;
}
```

The unsafe look-ups allow AES and other ciphers using S-boxes to be implemented in Hermes, but there is no guarantee of constant timing. It is possible to implement AES without using unsafe look-ups in S-boxes. For example, the unsafe look-up

```
x1 += unsafe S[x];
```

can be replaced by a sequence of safe conditional look-ups

```
for (i = 0; 256) {
  if (x==i) x1 += S[i];
  i++;
}
```

but this is obviously not very efficient. Another way around using a table look-up is computing $S[i]$ from scratch every time instead of caching it in a table – entries in the Rijndael S-box used in AES can be calculated using rotates and XORs – but that approach is also more expensive than table look-ups (in particular for the inverse matrix multiplication), though not as bad as the loop above. Implementing the mix-columns step using reversible logic operations is fairly fast, but the inverse (required for decryption) is relatively time-consuming to do without using tables.

Bernstein [10] suggests that S-boxes should be avoided when side-channel attacks may be an issue. We tend to agree, but nevertheless leave it as a possibility in Hermes.

We have implemented AES using S-boxes, mainly to show that it can be done in Hermes.

C code for the AES cipher shown in Figure B.19 is based on AES Dust [17], but modified slightly (which does not negatively affect performance). The `M0x3[]` parameter is a table for multiplication by 3 over GF(2^8). Multiplication by 2 is done using logic operations in the AddConstant step and by XORing multiplication by 3 (done with table lookup) with the value itself. We have not shown initialisation of the S-boxes and the multiplication table.

The Hermes version of the AES cipher is based on the C code, but has required extensive work to make it reversible. The code is shown in Figures B.17 and B.18. The part shown in Figure B.18 is mainly uncomputation of key expansion. In addition to the parameters for the C version, the Hermes version also needs the inverse of the S-box as well as multiplication tables for the larger constants used in the inverse mix-columns step. We have not shown initialisation of these. Many of the operations used in AES (such as key expansion and the mix-columns step) must be explicitly reversed in Hermes to reset local variables to zero. While this adds to the execution time, it allows using the same function in reverse for decryption, and it ensures that secret values are cleared.

5.5. Other implemented ciphers

We have implemented several other encryption algorithms in Hermes, including Red Pike [18] (a cipher used by GCHQ) and Blowfish [19] (designed as a replacement for DES).

6. Optimising Hermes with Partial Evaluation

For safety and reversibility, Hermes requires a number of run-time checks and assertions that equivalent C programs do not:

- Bounds check at array look-up.
- Verification that local variables and array are zeroed at the end of their scope.
- Verification that the counter variable of a loop does not return to its initial value after the first iteration.

This is bound to make Hermes slower than the equivalent C program, unless these checks can be optimised away. Array bounds checks can often be eliminated using standard data-flow techniques [20], but the other checks can be difficult to remove by simple static analysis.

One option is to assume the assertions hold and simply not do them. This is essentially the approach taken by C. This can be unsafe, but with thorough testing the probability that the checks will never fail can be made very low. Another possibility is to exploit that array sizes and indexes (except for unsafe look-ups) and counter variables are public data, and that public inputs (if any) are often fixed for any particular application: They typically represent sizes of keys and data blocks. Additionally, all control flow is public so timing does not depend on secret data. By fixing public inputs and array sizes to constants, we can employ partial evaluation [21, 22] to do all computations on public data (categorising these as static), leaving a straight-line residual program that only does computations on secret (dynamic) data. This will eliminate loop counters and replace all array sizes and safe array indexes by constants, so bounds checks for safe array lookups become trivial. Additionally, verifying that public local variables are zeroed at the end of their scope becomes trivial. The only run-time checks that are not removed are index checks for unsafe array lookups and checks that secret local variables are zeroed at the end of their scope but, in a program without branching, such verification is easier to do at compile time than similar verification on a program with non-trivial control flow. Additionally, as computations on public data are eliminated, the residual program will do fewer computations than the original. Furthermore, since there is no conditional branching, side-channel attacks that target weaknesses in speculative execution (such as Spectre [23]) are avoided.

Partial evaluation has already been done on the reversible language Janus [6, 24]. A complication in that work was that not all control could be eliminated during partial evaluation, and the residual control could not always be expressed using the structured control structures of Janus. Here, we can eliminate all control, so partial evaluation is much easier. Additionally, we do not need to do binding-time analysis, as the type system already distinguishes between static (public) and dynamic (secret) data. The approach is as follows:

1. Make two copies of every procedure: One for forwards execution and one for backwards execution. The latter are made by inverting the procedure bodies. Uncalls are changed to calls to the backwards versions of the procedures.
2. Adding a statement `assert` e`;` to the language, and reifying all checks as explicit assertions. In assertions, we allow a new expression form `allZero` a`[`n`];` that verifies that array a has size n and that all entries in the array are zero. This is used for reifying the checks done by $disposearray_z$.
3. Fix array sizes and public inputs (if any) to constant values.
4. Evaluate all public computations and emit all computations on secret values as residual code. Procedure calls are inlined, and their bodies partially evaluated. An unsafe array lookup a`[`i`]`, where i is an 8-bit variable and the size of a is at least 256 needs no index check, so these can be eliminated even if i is an unknown (secret) value. This is a common case for S-boxes.

When compiling the resulting residual programs, only the remaining assertions are compiled into run-time checks.

As an example, reifying the checks in the Speck128 procedure in Figure B.14 results in the program shown in Figure B.15 (with some parts omitted to fit on a page). We have added a suffix to the names of the inverted procedures. The main function has no public parameters, but the sizes of the arrays `ct` and `K` are public, so the procedure is specialised to these sizes (which are both 2). The only other public values are the loop bounds (0 and 32) and the loop counter `i`. The loop is unrolled, and call to Rp and Rs and their inverses are inlined and specialised. The residual code (somewhat abbreviated) can be seen in Figure B.16. Note that the only residual assertions are that the local variables are zeroed at the end of their scope.

7. Compilation and Benchmarks

We have made a prototype compiler for the subset of Hermes that is output by the partial evaluator, targeting x86-64 by using gcc inline assembler. The compiler is fairly straightforward and does no optimisation other than what the partial evaluator has done.

Having a compiler to x86-64 allows us to compare code size and timings of implementations in C and Hermes of various lightweight ciphers. Table 1 shows the results. Code sizes are in bytes for the .o file generated by gcc -c for a file containing the encryption function and required header file includes only. The laptop on which the benchmarks were run has an Core i7-6500U CPU running at 2.50GHz, has 8GB of RAM, and runs Ubuntu 16.04 LTS. Compilation was done using gcc version 5.4.0 for Ubuntu using the -O2 optimisation flag for the C programs but not the Hermes programs, as optimisation of the inline assembly code might compromise constant timing and clearing of registers. Running times are measured in seconds using the Linux `time` command and are for 10^8 calls (back-to-back) for the encryption function, except for AES which is for 10^7 calls. The numbers for RC5 exclude key expansion and is for the 12-round version of RC5.

Note that gcc recognises and generates rotate instructions for the rotate macros used by Speck128, RC5, and RedPike.

The Hermes programs are larger (sometimes considerably so), which is mainly due to the aggressive loop unrolling done by the partial evaluator. They are also mostly slower, which is mainly because Hermes does some clean up that C does not. For example, Hermes checks that variables are cleared at the end of their scope, which the C versions do not, and the Hermes versions of Speck 128 and RedPike uncompute the key expansions (to ensure reversibility). For AES and Speck 128, in particular, this uncomputation is significant.

In AES, Hermes needs (for reversibility) to uncompute the mix-columns step as well as the S-box look-ups and the key expansion, which adds a large overhead. AES has nested loops with bodies of significant size, so the loop unrolling

Program	code size	time	Hermes/C
TEA (C)	1464	11.29	
TEA (Hermes)	4440	11.30	1.00
Speck 128 (C)	1432	5.30	
Speck 128 (Hermes)	3304	8.83	1.67
RC5 (C)	1400	2.56	
RC5 (Hermes)	1888	2.98	1.16
RedPike (C)	1400	3.14	
RedPike (Hermes)	2160	3.51	1.12
AES (C)	2176	4.26	
AES (Hermes)	58824	13.85	3.25

Table 1: Comparison of C and Hermes

increases the code size dramatically, enough to cause instruction-cache misses even after the first execution of the encryption procedure. With the exception of AES, porting the ciphers from C to Hermes was relatively straightforward.

8. Conclusion and Future Work

We have presented a language Hermes for writing lightweight 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.

While this paper shows implementation of well-known ciphers in Hermes, we suggest that Hermes can also be used for developing new ciphers. By using Hermes, reversibility, avoidance of information leaks, and constant timing of secret operations are maintained as invariants during development of the cipher (as long as you avoid unsafe array lookups). This is similar to how statically typed languages can maintain type safety as a statically checked invariant. And, just like statically typed languages can sometimes feel restrictive, writing a cipher in Hermes can be more cumbersome than in C. In both cases, the safety provided by the restrictions can be worth the extra effort. Hermes does not protect against attacks against the cipher algorithm itself, such as differential attacks or exploitation of weak keys, but it can let the cipher developer concentrate on these aspects instead of lower-level insecurities.

Hermes has a formal semantics for both the type system and run-time behaviour. These semantics can be used to prove both that secret information does not leak into public variables and that type-correct programs are, indeed, reversible. In this paper, we have proven reversibility. It can, however, not be proven from the semantics alone that timing does not depend on secret values, as it relies on specifics of implementation on a particular platform. At best, we can argue informally that this will be the case if compilation does not introduce variable timing.

The semantic rules do not specify what happens if an assertion in a semantic rule fails, for example when an array bound is exceeded or a local variable is not cleared at the end of a block. For the type rules, the obvious behaviour is a compile-time 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. The latter will, however, compromise reversibility: Reversibility is only guaranteed if the semantic rules gives termination with no failed assertions. If partial evaluation is used as a preprocessor to compilation, all errors caused by operations on public values will be reported by the partial evaluator, leaving only errors caused by operations on secret values. Since the sequence of operations is not influenced by secret values, the amount of testing required to give a reasonable assurance that there will be no such errors is modest.

We have in Standard ML implemented a number of programs for executing Hermes:

- 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.

- An assertion reifier that turns all runtime tests into explicit assertions. It also adds inverted procedures, thereby eliminating `uncall`.

- A partial evaluator for assertion-reified Hermes that treats public data as known and secret data as unknown. This can be used to eliminate most of the run-time checks required by Hermes and, additionally, optimises the code at the cost of increasing code size (because loops are unrolled and procedure calls inlined).

- A prototype compiler from Hermes to x86-64 for the subset of Hermes that is output by the partial evaluator. The output from the compiler is a sequence of instructions with no branches (except for aborts at errors), and none of the issued instructions except memory references have non-constant timing, and since (with the exception of unsafe array lookups) the addresses accessed by memory references are independent of secret values, we argue that the compiled programs are free of timing-based side-channel attacks, provided they run to completion without error and do not use unsafe array lookups. Errors are reported by returning a non-zero value (encoding the position of the failing assert statement in the source code) in the RAX register.

The partial evaluator increases the code size, often dramatically. Since its main purpose is to remove unnecessary run-time tests, a possibility is to run the partial evaluator, inspect its output to identify the tests that have been removed, and then remove these from the original program. This will avoid the code explosion caused by the partial evaluator but still remove most of the run-time tests. This will, however, complicate the subsequent compilation.

We also have prototype compilers for earlier versions of Hermes to C and WebAssembly [25]. These can not guarantee against side channel attacks, as the compilers that compile C and WebAssembly to native code may not preserve constant time, and they may optimise away code that clears memory.

We have implementing a selection of lightweight ciphers including the Advanced Encryption Standard (AES) in Hermes. An issue with AES is that it uses secret information as array indexes to S-boxes, which may leak information about secret values due to cache timing. This was not allowed in the previous version of Hermes, so we have added unsafe array look-ups to the current version. Unsafe look-ups must, of course, be used with care and preferably avoided altogether. An if-then-else construct was also added to more easily implement AES.

While the speed of the simple ciphers when implemented using the Hermes compiler are similar to the speed of similar C implementations, AES was much slower and larger. This is partly due to the complexity of AES and partly due to use of S-boxes, which Hermes does not handle efficiently. So while implementing AES in Hermes is a proof of concept showing that it can be done, we do not recommend using Hermes for ciphers that employ S-boxes. Key expansion has to be uncomputed when using Hermes (to ensure reversibility), so we recommend that key expansion is done offline before multiple calls to the main cipher function rather than inline in every call. The difference can be seen in the timings of Speck 128 (which uses inline key expansion) and RC5 (which uses offline key expansion).

Public-key ciphers 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 cipher text, but also additional "garbage" information that must be discarded before transmitting the cipher 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] D. Táborský, K. F. Larsen, M. K. Thomsen, Encryption and reversible computations - work-in-progress paper, in: Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings, 2018, pp. 331–338. `doi:10.1007/978-3-319-99498-7\_23`. URL `https://doi.org/10.1007/978-3-319-99498-7_23`

[2] C. Lutz, Janus: a time-reversible language, A letter to Landauer. http:://www.tetsuo.jp/ref/janus.pdf (1986).
URL `http://www.tetsuo.jp/ref/janus.pdf`

[3] T. Yokoyama, H. B. Axelsen, R. Glück, Principles of a reversible programming language, in: Proceedings of the 5th conference on Computing frontiers, CF '08, ACM, New York, NY, USA, 2008, pp. 43–54.

doi:http://doi.acm.org/10.1145/1366230.1366239.
URL http://doi.acm.org/10.1145/1366230.1366239

[4] T. Æ. Mogensen, Hermes, a reversible language for writing encryption algorithms (work in progress), in: Proceedings of PSI 2019, Lecture Notes in Computer Science 11964, Springer Berlin Heidelberg, 2020, pp. 243–251.

[5] T. Æ. Mogensen, Hermes, a language for writing encryption algorithms, in: Proceedings of RC 2020, Lecture Notes in Computer Science 12227, Springer Berlin Heidelberg, 2020, pp. 93–110.

[6] J. Hatcliff, T. Æ. Mogensen, P. Thiemann (Eds.), Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, 1998, Vol. 1706 of Lecture Notes in Computer Science, Springer, 1999.
URL http://dblp.uni-trier.de/db/conf/pepm/pe1998.html

[7] J. Palsberg, P. Ørbæk, Trust in the λ-calculus, in: A. Mycroft (Ed.), Static Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 314–329.

[8] G. Smith, Principles of secure information flow analysis, in: M. Christodorescu, S. Jha, D. Maughan, D. Song, C. Wang (Eds.), Malware Detection, Springer US, Boston, MA, 2007, pp. 291–307.

[9] F. P. Miller, A. F. Vandome, J. McBrewster, Advanced Encryption Standard, Alpha Press, 2009.

[10] D. J. Bernstein, Cache-timing attacks on AES, Tech. rep. (2005).
URL https://pdfs.semanticscholar.org/fbf1/8eb234f19897f758fedeb69c52f29fbbf15a.pdf

[11] D. J. Wheeler, R. M. Needham, TEA, a tiny encryption algorithm, in: B. Preneel (Ed.), Fast Software Encryption, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 363–366.

[12] Wikipedia, Tiny encryption algorithm, https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm (Accessed January 2019).

[13] R. L. Rivest, The RC5 encryption algorithm, Dr. Dobb's Journal 20 (1) (1995) 146–148.

[14] Wikipedia, RC5, https://en.wikipedia.org/wiki/RC5 (Accessed February 2019).

[15] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers, The SIMON and SPECK families of lightweight block ciphers, Cryptology ePrint Archive, Report 2013/404, https://eprint.iacr.org/2013/404 (2013).

[16] Wikipedia, Speck (cipher), https://en.wikipedia.org/wiki/Speck_(cipher) (
Accessed February 2019).

[17] Odzhan, AES dust, https://github.com/odzhan/aes_dust (Accessed September 2020).

[18] Wikipedia, Red pike (cipher), https://en.wikipedia.org/wiki/Red_Pike_(cipher) (Accessed February 2019).

[19] B. Schneier, Description of a new variable-length key, 64-bit block cipher (Blowfish), in: R. Anderson (Ed.), Fast Software Encryption, Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, pp. 191–204.

[20] T. Æ. Mogensen, Introduction to Compiler Design (2nd Edition), Springer International Publishing, Berlin, 2017.

[21] N. D. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[22] J. Hatcliff, T. Mogensen, P. T. (Eds.), Partial Evaluation: Practice and Theory, Vol. 1706 of Lecture Notes in Computer Science, Springer Verlag, 1999.

[23] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution., meltdownattack.com.

[24] T. Æ. Mogensen, Partial evaluation of the reversible language Janus., in: S.-C. Khoo, J. G. Siek (Eds.), PEPM 2011, ACM, 2011, pp. 23–32. URL `http://dblp.uni-trier.de/db/conf/pepm/pepm2011.html#Mogensen11`

[25] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien, Bringing the web up to speed with webassembly, SIGPLAN Not. 52 (6) (2017) 185–200. doi:10.1145/3140587.3062363.
URL `http://doi.acm.org/10.1145/3140587.3062363`

Appendix A. Proof of Reversibility

We prove that Hermes is a reversible language by proving that, for any statement s, the statement sequence $s\,I(s)$ has no net effect, where I is the statement inversion function shown in Figure 7. More precisely, we will prove that if $\Delta, \eta \models_S s : \sigma_1 \rightleftharpoons \sigma_2$, then $\Delta, \eta \models_S I(s) : \sigma_2 \rightleftharpoons \sigma_1$. Note that the prerequisite is that s terminates without error, since executing a non-terminating statement or a statement that terminates in error followed by its inverse is obviously not the same as doing nothing.

The proof uses the semantic rules and the restrictions imposed by the type system, but is mainly informal.

Empty statement We have $I(;) = ;$ and $\Delta, \eta \models_S\ ; : \sigma \rightleftharpoons \sigma$, so it is trivial that the inverse also terminates and the composition of the two has no net effect.

Update We note that the expression e does not contain the root variable on the left-hand side of the update. This means that the location that the left-hand side l evaluates to is not accessed when evaluating e. Hence, in a sequence $l \oplus\!=_1\; e\texttt{;}\; l \oplus\!=_2\; e\texttt{;}$, e evaluates to the same in both updates. Also, l evaluates to the same sized location (z, λ) both times, as any index expression used in l is unaffected by the updates. So, if the value stored at λ before the sequence is v_1 and e evaluates to v_2, the net effect is to update λ to contain $U(\oplus\!=_2\; (U(\oplus\!=_1\; (v_1, v_2 \downarrow_z)), v_2 \downarrow_z))$. We also note that if the first update evaluates without error, so will the second.

If $\oplus\!=_1$ is `^=`, $\oplus\!=_2$ is also `^=`, so $U(\oplus\!=_2\; (U(\oplus\!=_1\; (v_1, v_2 \downarrow_z)), v_2 \downarrow_z))$ reduces to $(v_1 \,\texttt{\^{}}\, (v_2 \downarrow_z)) \,\texttt{\^{}}\, (v_2 \downarrow_z)$, where `^` is bit-wise exclusive OR. By associativity, we get $v_1 \,\texttt{\^{}}\, ((v_2 \downarrow_z) \,\texttt{\^{}}\, (v_2 \downarrow_z))$, which is equal to $v_1 \,\texttt{\^{}}\, 0$, which is equal to v_1, which was the original value of λ.

If $\oplus\!=_1$ is `+=`, $\oplus\!=_2$ is `-=`, so we by similar reasoning get $(v_1 \,\texttt{+}\, (v_2 \downarrow_z)) \,\texttt{-}\, (v_2 \downarrow_z)$, which is equal to v_1 (since we operate modulo 2^z). The case for `-=` is similar.

Swap A swap statement reverses to itself.

The restrictions on swap statements ensure that the two locations λ_1 and λ_2 are different, and that neither is used when calculating the L-values. So in the sequence $l_1\texttt{<->}l_2\texttt{;}\, l_1\texttt{<->}l_2\texttt{;}$, the same locations are calculated in both swap statements. If executing the first swap produces no error, neither will the second. The net effect of swapping the contents of two locations twice is nothing.

CondSwap A conditional swap statement reverses to itself.

By the restrictions, neither the evaluation of e to v, nor the evaluation of the two L-values to locations λ_1 and λ_2 uses the contents of λ_1 and λ_2, so updating λ_1 and λ_2 does not change what e or the two L-values evaluate to in the second conditional swap statement. In particular, the value of the condition is not changed by the swap, and if it evaluates without error the first time, it will also do so the second time.

If the condition e evaluates to 0, the net effect of the conditional swap is nothing (rule *CondSwap*1), so doing this twice also has no effect. If e evaluates to a non-zero value, the net effect of the conditional swap is the same as for an unconditional swap, so doing it twice has no net effect.

If-then-else An if-then-else statement is reversed by inverting the branches. The restrictions on updates in the branches ensures that the value of the condition e is not changed by executing a branch. This includes that if e evaluates without error in the original statement, it will do so also in the reversed statement.

If e evaluates to a non-zero value, the net effect of executing the if-then-else statement followed by its inverse is that of executing the first branch followed by the inverse of that. By induction, we get that this has no net effect. If e evaluates to 0, the net effect is executing the second branch followed by its inverse, which by induction has no net effect.

For loop We assume that the for loop terminates without error, as we otherwise have nothing to prove. The simple case is that the initial and final bounds are equal, which implies that the body is never executed, so by rule *Loop*1, the store is unchanged by the loop body. The only other changes to the store are creating a location λ for the loop counter x, giving λ the value of the initial (and final) loop bound, resetting λ to zero, and then disposing of it. By the informal definitions of $newlocation_{64}$ and $disposelocation_{64}$, the net effect of this is no effect.

If the bounds are different, we can unroll the loop into a block of the form

```
{public u64 $x$, $v_1$, $v_2$;
 $v_1$+=$e_1$; $v_2$+=$e_2$; $x$+=$v_1$;
   /*$x$==$v_1$*/
   /*$x$!=$v_2$*/  $s$  /*$x$!=$v_1$*/
 ...
   /*$x$!=$v_2$*/  $s$  /*$x$!=$v_1$*/
   /*$x$==$v_2$*/
 $x$-=$v_2$; $v_2$-=$e_2$; $v_1$-=$e_1$;}
```

where s is repeated the number of iterations and v_1 and v_2 are variables not occurring elsewhere. Note that between iterations, x is (by the semantic rule *Loop*2) different from both v_1 and v_2, which we have asserted in comments.

The only operational differences between the unrolled loop and the original loop is that the tests on the loop counter are eliminated and that we hold v_1 and v_2 in explicit variables that are created at the start of the block and disposed of at the end of the block, where the semantic rules store v_1 and v_2 directly in the rules. But the net effect is the same. The restriction that variables influencing the bounds expressions can not be modified inside the loop ensures that the bounds expressions have the same value after the loop as before it is entered, so the subtractions at the end of the unrolled loop clears the variables to zero.

Reversing this block gives us

```
{public u64 x, v1, v2;
 v1+=e1; v2+=e2; x+=v2;
  /*x==v2*/
  /*x!=v1*/  I(s)  /*x!=v2*/
 ...
  /*x!=v1*/  I(s)  /*x!=v2*/
  /*x==v1*/
 x-=v1; v2-=e2; v1-=e1;}
```

The restriction that x can only be equal to v_1 at the start of the loop and only equal to v_2 at the end of the loop ensures that the number of occurrences of $I(s)$ in the reversed block is equal to the number of iterations of the reversed loop, since we, by induction, know that $I(s)$ undoes the effect of s. So the reversed block is an unrolling of the reversed loop. By induction, if the body statements do not produce errors in the original loop, nor will they in the reversed loop.

Call and Uncall These are the inverses of each other. We note that the environments created in the rules *Call* and *Uncall* are the same, and the store is modified by s and $I(s)$, respectively, in the same environment. And by induction, if s terminates without error, so does $I(s)$, and $I(s)$ undoes the effects of s, and since $I(I(s)) = s$, s undoes the effects of $I(s)$. So a call followed by an uncall to the same function with the same parameters has no net effect, and the same holds for a call followed by an uncall.

Block We need to show several things:

1. If $\eta, \sigma \models_D d \rightsquigarrow \eta', \sigma_0$, then $\sigma_0(\lambda) = \sigma(\lambda)$ for all locations λ in η, and σ_0 binds all λ that are in η' and not in η to either 0 or arrays with all elements bound to 0.
2. If $\eta', \sigma_0 \models_D^{inv} d \rightsquigarrow \eta, \sigma_{n+1}$, then $\sigma_n(\lambda) = \sigma_{n+1}(\lambda)$ for all locations λ in η, and σ_n binds all λ that are in η_n and not in η to either 0 or arrays with all elements bound to 0.
3. Successfully disposing of variables and arrays and then creating the same variables and arrays has no net effect.

For point 1, we observe that the rules for $\models_D$ only modify stores with *newlocation* and *newarray*, and that these affect only new locations (not in η) and (by the informal definitions of *newlocation* and *newarray*) bind these to 0 or arrays with all elements bound to 0. Similarly for point 2, the rules for $\models_D^{inv}$ only modify stores with *disposelocation* and *disposearray*, and that these do not affect locations in η_{n+1}, and (by their informal definitions) would give errors if the disposed locations are not cleared to 0 or zero-cleared arrays.

Point 3 assumes that the block finishes without errors: Successfully disposing variables and arrays by *disposelocation* and *disposearray* implies that these must have been zero-cleared before disposal. And creating them again will make then zero-cleared again, and since they are created in the opposite order of their disposal, the stack-like behaviour of allocations and disposals implies that the same locations are reused. Se the discussion at the beginning of Section 4.

So the statement sequence $\{d\ s_1\ \ldots\ s_n\}\ \{d\ I(s_n)\ \ldots\ I(s_1)\}$ (the block followed by its inverse) will create zero-cleared variables and arrays declared in d, execute $s_1\ \ldots\ s_n$, dispose the variables and arrays in d, creating these again, then executing $I(s_n)\ \ldots\ I(s_1)$, and finally disposing the variables and arrays in d once again. By point 3, this is equivalent to the single block $\{d\ s_1\ \ldots\ s_n\ I(s_n)\ \ldots\ I(s_1)\}$. Since the sequence $s_1\ \ldots\ s_n\ I(s_n)\ \ldots\ I(s_1)$ (by induction) has no net effect, the single block is equivalent to $\{d\ \}$. And by point 1 and 2, this has no net effect on the store.

This concludes the proof of reversibility.

Appendix B. Code Examples

We have collected the figures that show Hermes and C code examples in this appendix.

```
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++;
    }
  }
}
```

```
#define ROTL(X,R) (((X)<<((R)&31))|((X)>>(32-((R)&31))))
#define r 12

void encrypt(uint32_t pt[], uint32_t S[])
{
   uint32_t i, A = pt[0] + S[0], B = pt[1] + S[1];

   for (i = 1; i <= r; i++)
   {
      A = ROTL(A ^ B, B) + S[2*i];
      B = ROTL(B ^ A, A) + S[2*i + 1];
   }
   pt[0] = A; pt[1] = B;
}
```

Figure B.12: RC5 core in Hermes (top) and C (bottom)

```
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]; @
    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 */
}
```

```
void encrypt (uint32_t v[2], uint32_t k[4]) {
    uint32_t v0 = v[0], v1 = v[1], sum=0, i;
    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], i;
    uint32_t delta = 0x9E3779B9,          /* key schedule constant */
             sum = 0xC6EF3720;            /* sum = 32*delta */
    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;
}
```

Figure B.13: TEA in Hermes (top) and C (bottom)

```
#define ROR(x, r) ((x >> r) | (x << (64 - r)))
#define ROL(x, r) ((x << r) | (x >> (64 - r)))
#define R(x, y, k)\
  (x = ROR(x, 8), x += y, x ^= k, y = ROL(y, 3), y ^= x)
#define ROUNDS 32

void encrypt(uint64_t ct[2], uint64_t const K[2])
{
   uint64_t y = ct[0], x = ct[1], b = K[0], a = K[1];

   R(x, y, b);
   for (int i = 0; i < ROUNDS - 1; i++) {
      R(a, b, i); /* key expansion */
      R(x, y, b);
   }

   ct[0] = y;
   ct[1] = x;
}
```

```
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);
     }
   }
}

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 B.14: Speck128 in C (top) and Hermes (bottom)

```
speck128(u64 ct[], u64 K[])
{
   u64 y, x, b, a;
   assert 0 < size ct; y <-> ct[0];
   assert 1 < size ct; x <-> ct[1];
   assert 0 < size K; b += K[0];
   assert 1 < size K; a -
  += K[1];
   call Rs(x, y, b);
   for (i=0; 31) {
     call Rp(a, b, i); i++;
     call Rs(x, y, b); assert i != 0;
   }
   for (i=31; 0) {
     i--; call Rp_I(a, b, i); assert i != 31;
   }
   assert 0 < size ct; y <-> ct[0];
   assert 1 < size ct; x <-> ct[1];
   assert 0 < size K; b -= K[0];
   assert 1 < size K; a -= K[1];
   assert a==0; assert b==0; assert x==0; assert y==0;
}

speck128_I(u64 ct[], u64 K[])
{
   /* Omitted for brevity -- not needed for specialization */
}

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

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

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

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

Figure B.15: Speck128 with reified checks (abbreviated)

```
speck128_2_2(u64 ct[], u64 K[])
{
   u64 y, x, b, a;
   y <-> ct[0]; x <-> ct[1]; b <-> K[0]; a <-> K[1];

   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }

   { a >>= 8; a += b;           b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }
   { a >>= 8; a += b; a ^= 1; b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }
   { a >>= 8; a += b; a ^= 2; b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }

   . . . /* 50 lines omitted */

   { a >>= 8; a += b; a ^= 28; b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }
   { a >>= 8; a += b; a ^= 29; b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }
   { a >>= 8; a += b; a ^= 30; b <<= 3; b ^= a; }
   { x >>= 8; x += y; x ^= b; y <<= 3; y ^= x; }

   { b ^= a; b >>= 3; a ^= 30; a -= b;   a <<= 8; }
   { b ^= a; b >>= 3; a ^= 29; a -= b;   a <<= 8; }
   { b ^= a; b >>= 3; a ^= 28; a -= b;   a <<= 8; }

   . . . /* 25 lines omitted */

   { b ^= a; b >>= 3; a ^= 2; a -= b;   a <<= 8; }
   { b ^= a; b >>= 3; a ^= 1; a -= b;   a <<= 8; }
   { b ^= a; b >>= 3;          a -= b;   a <<= 8; }

   y <-> ct[0]; x <-> ct[1]; b <-> K[0]; a <-> K[1];
   assert a==0; assert b==0; assert x==0; assert y==0;
}
```

Figure B.16: Residual Speck128 program (somewhat abbreviated)

```
aes_ecb(u8 mk[], u8 s[], u8 sbox[], u8 sboxInv[],
        u8 M0x3[], u8 M0xe[], u8 M0xb[], u8 M0xd[], u8 M0x9[]) {
  u8 t, x[16], k[16]; public u8 tp, rc;
  { rc ++; for(i=0; 16) { k[i] ^= mk[i]; i++; } } @
  { // copy 28-bit master key to x
    for(i=0; 16) { x[i] <-> s[i]; i++; }
    for(round = 0; 11) { // AddRoundKey
      for(i = 0; 16) { s[i] <-> x[i]; s[i] ^= k[i]; i++; }
      if(round != 10) {  // if last round, stop
        k[0] ^= rc; rc <<= 1; // AddConstant
        { tp += rc & 1; @ if (tp) rc ^= 0x1a; }
        for(i=0; 4) { // ExpandKey
          t ^= k[12+((i-3)&3)]; @ k[i] ^= unsafe sbox[t];
          i++;
        }
        for(i=0; 12) {
          t ^= k[i]; @ k[i+4] ^= t;
          i++;
        }
        for (i=0; 16) { // SubBytes and ShiftRows
          x[(13*i)&15] ^= unsafe sbox[s[i]];
          s[i] ^= unsafe sboxInv[x[(13*i)&15]];
          i++;
        }
        // if not last round
        if(round!=9) { // MixColumns
          for(i=0; 16) {
            u8 a, b, c, d;
            a ^= x[i]; b ^= x[i+1]; c ^= x[i+2]; d ^= x[i+3];
            x[i] ^= unsafe M0x3[a^b] ^ c ^ d;
            x[i+1] ^= unsafe M0x3[b^c] ^ a ^ d;
            x[i+2] ^= unsafe M0x3[c^d] ^ a ^ b;
            x[i+3] ^= unsafe M0x3[a^d] ^ b ^ c;
            a ^= unsafe M0xe[x[i]] ^ unsafe M0xb[x[i+1]] ^
                 unsafe M0xd[x[i+2]] ^ unsafe M0x9[x[i+3]];
            b ^= unsafe M0x9[x[i]] ^ unsafe M0xe[x[i+1]] ^
                 unsafe M0xb[x[i+2]] ^ unsafe M0xd[x[i+3]];
            c ^= unsafe M0xd[x[i]] ^ unsafe M0x9[x[i+1]] ^
                 unsafe M0xe[x[i+2]] ^ unsafe M0xb[x[i+3]];
            d ^= unsafe M0xb[x[i]] ^ unsafe M0xd[x[i+1]] ^
                 unsafe M0x9[x[i+2]] ^ unsafe M0xe[x[i+3]];
            i += 4;
          }
        } else ;
      } else ;
      round++;
    }
```

Figure B.17: Hermes code for AES (part 1)

```
    for(round = 11; 0) {
      round--;
      if(round != 10) {
        for(i=12; 0) {
          i--;
          t ^= k[i]; @ k[i+4] ^= t;
        }
        for(i=4; 0) {
          i--;
          t ^= k[12+((i-3)&3)]; @ k[i] ^= unsafe sbox[t];
        }
        { tp += rc&1; @ if (tp) rc ^= 0x1a; }
        rc >>= 1; k[0] ^= rc;
      } else ;
    }
  }
}
```

Figure B.18: Hermes code for AES (part 2)

```
void aes_ecb(void *mk, void *data, uint_8t sbox[], uint_8t M0x3[]) {
  uint_8t a,b,c,d,i,j,t,w,x[16],k[16],rc=1,*s=(uint_8t*)data;

  // copy 128-bit plain text + 128-bit master key to x
  for(i=0;i<16;i++)  x[i]=s[i], k[i]=((uint_8t*)mk)[i];
  for(;;) {
    for(i=0;i<16;i++) s[i]=x[i]^k[i];  // AddRoundKey
    if(rc==108) break; // if last round, stop
    k[0]^=rc; rc=(rc<<1)^((-(rc>>7))&0x1b);  // AddConstant
    // ExpandKey
    for(i=0;i<4;i++) k[i]^=sbox[k[12+((i-3)&3)]];
    for(i=0;i<12;i++) k[i+4]^=k[i];
    for(w=i=0;i<16;i++) // SubBytes and ShiftRows
      ((uint_8t*)x)[w]=sbox[((uint_8t*)s)[i]],w=(w-3)&15;
    if(rc!=108) { // if not last round
      // MixColumns
      for(i=0;i<16;i+=4) {
        a=x[i],b=x[i+1],c=x[i+2],d=x[i+3];
          x[i] ^= M0x3[a^b] ^ c ^ d;
          x[i+1] ^= M0x3[b^c] ^ a ^ d;
          x[i+2] ^= M0x3[c^d] ^ a ^ b;
          x[i+3] ^= M0x3[a^d] ^ b ^ c;
      }
    }
  }
}
```

Figure B.19: C code for AES