Hermes
A reversible language for lightweight encryption
Mogensen, Torben Ægidius

Published in:
Science of Computer Programming

DOI:
10.1016/j.scico.2021.102746

Publication date:
2022

Document version
Peer reviewed version

Document license:
CC BY-NC-ND

Citation for published version (APA):
Hermes: A Reversible Language for Lightweight Encryption

Torben Ægidius Mogensen

DIKU, University of Copenhagen
Universitetsparken 5, DK-2100 Copenhagen Ø, 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, by ensuring no secret values are left in memory and 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

Light-weight encryption algorithms, such as RC5 [1], speck [2] and AES [3] are block ciphers that are implemented using low-cost operations such as bitwise logical operations, making encryption of a block fast even on small microprocessors as used in embedded systems. In many applications, light-weight encryption must be robust against side-channel attacks, most notably timing attacks and memory inspection. So the algorithms used and their implementations must not leave secret data in uncleared stack frames and similar places, and the time used to encrypt a block must not depend on the values of secret data, i.e. the
encryption key and the data that is to be encrypted.

Lightweight ciphers tend to be symmetric: Decryption and encryption and uses the same key and are computationally inverse of each other: The steps used during encryption are reversed and inverted for decryption. This makes it natural to explore using reversible programming languages for lightweight encryption. RC5, speck, AES and the other ciphers used in this article all use the same key for encryption and decryption, and with the exception of key expansion and the matrix multiplication in AES (more about this later), decryption is a fairly straightforward inversion of encryption, doing reverse steps in reverse order.

**Related work**

Recent work [4] has investigated using the reversible language Janus [5, 6, 7] 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. Additionally, Janus 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.

Janus is, however, not robust against timing-based attacks: It has operations and control structures the timing of which depend on the values of variables, and it does not distinguish between secret and public values, so ensuring that secret values do not affect control can require tedious analysis.

We propose a new reversible language, Hermes, specifically designed to address these concerns by clearly separating secret and public data and restricting operations on secret data to those whose time do not depend on the actual values processed. Although inspired by Janus, Hermes has some significant differences, as we shall see below.

Earlier and simpler versions of the Hermes language were presented at PSI’19 [8]
Working with these early versions of Hermes have prompted both extensions and restrictions to the language, notably a type system that separates secret and public data, which was added to the 2020 version. This distinction is also found in the type system for CT-Wasm [10], which is a variant of the Web assembly [11] intended to avoid side-channel attacks by restricting operations on secret values much in the same way as we do in Hermes. The Hermes type system not only tracks flow of information similar to binding-time analysis [12, 13, 14], trust analysis [15], and information flow analysis [16] but also imposes restrictions to ensure reversibility and (under reasonable assumptions) avoid timing-based side-channel attacks. Compared to CT-Wasm, Hermes is a higher-level language and adds reversibility. Note that we (like CT-Wasm) aim to prevent timing-based attacks by restricting operations on secret values, not by padding programs with extra computations that hide fluctuations in timing caused by operations on secret values [17, 18].

New contributions

Working with the 2020 version of Hermes, including using Hermes to implement the Advanced Encryption Standard (AES) [19], 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 takes different time), but AES does, in fact, use secret values to index arrays (S-boxes). This is a known weakness of straightforward implementations of AES and other encryption methods that use S-boxes [20]. To allow AES and other encryptions using S-boxes to be implemented in Hermes, we have added a keyword unsafe to array lookups, 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. Other extensions include a reversible if-then-else construct (with public control), where a loop was the only control-flow structure in earlier versions.

Additionally, previous implementations of Hermes target high-level languages
or virtual machines, so they lack strong guarantees on time-invariance of operations on secret values. A new compiler targets x86-64 directly, so we have full control of instruction selection. The compiler exploits the public/secret type system also for optimisation: Public data is for nearly all light-weight encryption algorithms limited to the sizes of keys and blocks of data, so it is likely to remain constant for any given application. So we can use partial evaluation [12, 13], to do all computations on public data (categorising these as static), which removes a significant number of operations including most array index checks.

The Hermes language

We introduce the syntax of Hermes in Section 2. In Section 3, we show its type system, and in Section 4, we show its runtime semantics. We show some examples of Hermes programs in Section 5, in Section 6, we show how partial evaluation is used to optimise Hermes programs before compilation. We compare in Section 7 output from our Hermes compiler with output from a C compiler in terms of object code size and speed. We summarise results and possible future work in Section 9.

In appendices, we provide proofs of reversibility (in Appendix A) and non-interference [21]: that secrets do not leak to public variables or through timing (in Appendix B).

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.

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
Program $\rightarrow$ Procedure$^+$

Procedure $\rightarrow$ id (Args) Stat

Args $\rightarrow$ Type id | Type id[] | Args , Args

Type $\rightarrow$ secret IntType | public IntType

Stat $\rightarrow$ ;
  $|$ Lval update Exp ;  $|$ Lval <-> Lval
  $|$ if (Exp) Lval <-> Lval
  $|$ if (Exp) Stat else Stat
  $|$ for (id = Exp ; Exp) Stat
  $|$ call id (Lvals) ;  $|$ uncall id (Lvals);
  $|$ { Decls Stat$^*$}

Exp $\rightarrow$ Lval | numConst | size id
  $|$ Exp binOp Exp | unOp Exp

Lval $\rightarrow$ id | id [Exp] | unsafe id [Exp]

Lvals $\rightarrow$ Lval | Lval , Lvals

VarSpec $\rightarrow$ id | id [Exp]

Decls $\rightarrow$ Type VarSpec ; Decls
  $|$ const id = numConst ; Decls

Figure 1: Core syntax of Hermes
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 a binary 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. Using all ones for true makes it more convenient with bitwise logical operations, which in Hermes are &, 1, and ^.

**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 rotation. 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 >>= differs from their meaning in C, where they represent shift-updates (which are not reversible).

### 2.1. Example: Red Pike

To illustrate Hermes, we use Red Pike [22], a relatively simple cipher used by GCHQ, and compare implementations in C and Hermes. Both versions are shown in Figure [2].

We have derived the Hermes version directly from the C version without major modifications. Apart from minor syntactic differences, the main differences are

- The C version uses macros for rotating left and right, where the Hermes version uses the built-in update operators.
```c
#define CONST 0x9E3779B9
#define ROUNDS 16

#define ROIL(X, R) (((X) << ((R) & 31)) | ((X) >> (32 - ((R) & 31))))
#define ROIR(X, R) (((X) >> ((R) & 31)) | ((X) << (32 - ((R) & 31))))

void encrypt(uint32 *x, const uint32 *k) {
  uint32 k0 = k[0]; uint32 k1 = k[1];
  uint32 x0 = x[0]; uint32 x1 = x[1];
  for (uint32 i = 0; i < ROUNDS; i++) {
    k0 += CONST; k1 -= CONST;
    x0 ^= k0; x0 += x1; x0 = ROIL(x0, x1);
    x1 = ROIR(x1, x0); x1 -= x0; x1 ^= k1;
  }
  x[0] = x1; x[1] = x0;
}
```

Figure 2: Red Pike in C (top) and Hermes (bottom)
• When copying array elements to and from local variables, the C version uses assignments (\(=\)), where the Hermes version uses reversible updates and swaps.

• The contents of the local variables are cleared to 0 in the Hermes version. For \(x_0\) and \(x_1\), we do this by swapping with \(x[1]\) and \(x[0]\) that were cleared to 0 using the swaps at the beginning of the procedure. We generally use swaps to copy the message to and from local variables, as the message is changed by the procedure body. For keys \(k_0\) and \(k_1\), we use \(+=\) to initialise the local copies and \(-=\) to reset them to zero, as this is a bit cheaper.

Resetting the local variables to zero is required for reversibility, and ensures that, after execution, the only copies of secret values are in the parameter arrays. It would not be difficult to explicitly clear local variables in the C version, but there is a risk that the C compiler will remove such assignments to dead variables.

In the Hermes program, the loop counter \(i\) is implicitly public (all loops counters are), and all other variables are implicitly secret (as they are not declared public).

We have only specified the encryption function in C, where Hermes implicitly (through reversibility) specifies both encryption and decryption.

### 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 \texttt{constant}, which is implicitly a public 64-bit number. Note that \texttt{VarType} uses a special type \texttt{constant} for constant identifiers. This signals that you cannot update these identifiers. When used in expressions, \texttt{VarTypes} are converted to \texttt{ValTypes}, which are implicitly 64 bits. Constants are converted to public values. When used in updates, \texttt{VarTypes} are not converted, but constants cannot be used. This is indicated in the (Update) and (Swap) rules by requiring \texttt{VarTypes} to have sizes.
So we have:

\[
\begin{align*}
\text{ValType} &\rightarrow \text{secret} \mid \text{public} \\
\text{VarType} &\rightarrow \text{constant} \mid \text{ValType}^{\text{Size}} \mid \text{ValType}^{\text{Size}}[
\end{align*}
\]

\[
\text{Size} \rightarrow 8 \mid 16 \mid 32 \mid 64
\]

We use \( t \) with optional subscript to denote a value type, \( \tau \) 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 \( \text{public} \sqsubseteq \text{secret} \) 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 \( \rho \) 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 \( \rho \). We update environments using the notation \( \rho[x \mapsto \tau] \), which creates a new environment that is identical to \( \rho \), except that \( x \) is bound to \( \tau \).

Judgements for typing expressions (denoted by \( e \) with optional subscript), are of the form \( \rho \vdash E : \text{ValType} \), and judgements for typing L-values (denoted by \( l \) with optional subscript) are of the form \( \rho \vdash_L l : \text{VarType} \). 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 L-value or expression, \( R() \) finds the root variable of an L-value, and \( V_I() \) finds the variables in index expressions in an L-value. In the equations below (and later in type rules and semantics rules), \( \odot \) is a meta variable representing any binary operator, and \( \neg \) is a meta variable representing any unary operator (though, at present, Hermes supports only one: \( ~ \)). We will later also use \( \oplus \) as a meta variable representing any reversible update operator.
\[
V(x) = \{x\} \quad V(x[e]) = \{x\} \cup V(e)
\]
\[
V(\text{unsafe } x[e]) = \{x\} \cup V(e)
\]
\[
V(n) = \emptyset \quad V(\text{size } x) = \emptyset
\]
\[
V(e_1 \odot e_2) = V(e_1) \cup V(e_2) \quad V(\neg e) = V(e)
\]
\[
R(x) = x \quad V_I(x) = \emptyset
\]
\[
R(x[e]) = x \quad V_I(x[e]) = V(e)
\]
\[
R(\text{unsafe } x[e]) = x \quad V_I(\text{unsafe } x[e]) = V(e)
\]

The aliasing rules are the same as used in Janus, so we refer to the Janus literature\[6\] for their justification.

Note that \(V()\) does not include variables in \text{size}-expressions, as these are harmless in terms of aliasing. This makes \(V()\) different from the typical free-variables function.

We specify type rules for L-values and expressions in Figure 3.

For L-values, rule (Variable) states that a variable has the type specified by the environment. Rule (ArrayAccess) 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 lookups (Rule (UnsafeArrayAccess)) can use both public and secret indexes, but the elements of the array must be secret. This is because the result of lookups and updates may depend on secret values.

Rules (Constant1) and (Constant2) state that a constant is public. \(n\) denotes an integer constant. Rule (L-val) says 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. Rule (UnOp) just say that the result of an unary operator has the same type as its argument.

The rules for a binary operator \(\odot\) is more complex. If both arguments are
\[
\frac{\rho \vdash L \ x : \rho(x)}{\rho \vdash L \ x : \rho(x)} \text{(Variable)}
\]

\[
\frac{\rho(x) = t^\square \quad \rho \vdash E \ e : \text{public}}{\rho \vdash L \ x[e] : t^\square} \text{(ArrayAccess)}
\]

\[
\frac{\rho(x) = \text{secret}^\square \quad \rho \vdash E \ e : t}{\rho \vdash L \ \text{unsafe} \ x[e] : \text{secret}^\square} \text{(UnsafeArrayAccess)}
\]

\[
\frac{\rho \vdash E \ n : \text{public}}{\rho \vdash L \ l : \text{constant}^1} \quad \frac{\rho \vdash E \ l : \text{public}}{\rho \vdash L \ l : \text{constant}^2}
\]

\[
\frac{\rho \vdash E \ l : t^\square \quad \rho \vdash L \ l : t^\square}{\rho \vdash E \ l : t^\square} \text{(L-val)}
\]

\[
\frac{\rho \vdash E \ e : t}{\rho \vdash E \ e : t} \text{(UnOp)}
\]

\[
\frac{\rho \vdash E \ e_1 : t_1 \quad \rho \vdash E \ e_2 : t_2 \quad t_1 \sqcup t_2 = \text{public}}{\rho \vdash E \ e_1 \odot e_2 : \text{public}} \text{(BinOp1)}
\]

\[
\frac{\rho \vdash E \ e_1 : t_1 \quad \rho \vdash E \ e_2 : t_2 \quad t_1 \sqcup t_2 = \text{secret} \quad \odot \notin TV}{\rho \vdash E \ e_1 \odot e_2 : \text{secret}} \text{(BinOp2)}
\]

\[
\frac{\rho \vdash E \ e_1 : t_1 \quad \rho \vdash E \ e_2 : t_2 \quad t_1 \sqcup t_2 = \text{secret}}{\rho \vdash E \ e_1 \odot e_2 : \text{secret}}
\]

\[
\frac{\rho \vdash E \ e_1 : t_1 \quad \rho \vdash E \ e_2 : t_2 \quad t_1 \sqcup t_2 = \text{secret}}{\rho \vdash E \ e_1 \odot e_2 : \text{secret}}
\]

\[
\frac{\rho \vdash E \ size x : \text{public}}{\rho \vdash E \ size x : \text{public}} \text{(Size)}
\]

Figure 3: Type rules for L-values and expressions

public (rule (BinOp1)), the result is public. If any of the arguments are secret
(rule (BinOp2)), 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.

Rule (Size) states that the size of an array is a public value. We have chosen
that all array sizes are public (even if their contents are secret) because we want
index range checks to be entirely on public data.
3.2. Statements and local declarations

A judgement for a statement $s$ is of the form $\Gamma, \rho \vdash s$ and states that given a procedure environment $\Gamma$ and variable environment $\rho$, 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 4.

Rule (Empty) says that the empty statement is well typed. To ensure reversibility rule (Update) (where $\oplus$ denotes an update operator) says that the root variable of the L-value cannot occur in the expression nor in an index expression of the L-value. For example, the updates $x \leftarrow x;$, $a[i] <<= a[j];$, and $a[a[i]] += 1;$ are disallowed. Furthermore, if the expression is secret, the L-value must also be secret.

Rule (Swap) states that the two L-values must have exactly the same type, and that the root variable of one side cannot occur in index expressions on the other side. A swap of two elements of the same array $a[i] <-> a[j];$ is, however, allowed.

Rule (SwapC) 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

\[
\begin{align*}
\text{tmp} &= c \& (x \lor y); \\
x &= \text{tmp}; \\
y &= \text{tmp}; \\
\text{tmp} &= 0;
\end{align*}
\]

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

The condition in the more general if-then-else statement (rule (IfElse)) 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 runtime 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

$$
\begin{align*}
U(;) & = \emptyset \\
U(l_1 := e;) & = \{R(l)\} \\
U(l_1 \leftarrow l_2;) & = \{R(l_1), R(l_2)\} \\
U(\text{if } (c) \ l_1 \leftarrow l_2;) & = \{R(l_1), R(l_2)\} \\
U(\text{if } (c) \ s_1 \ \text{else } s_2) & = U(s_1) \cup U(s_2) \\
U(\text{for } (x:=e_1; e_2) \ s) & = U(s) \setminus \{x\} \\
U(\text{call } f(l_1, \ldots, l_n);) & = \{R(l_1), \ldots, R(l_n)\} \\
U(\text{uncall } f(l_1, \ldots, l_n);) & = \{R(l_1), \ldots, R(l_n)\} \\
U(\{d \ s_1 \ldots s_n\}) & = (U(s_1) \cup \cdots \cup U(s_n)) \setminus D(d)
\end{align*}
$$

$$
\begin{align*}
D() & = \emptyset \\
D(t \ \text{uz } x; \ d) & = \{x\} \cup D(d) \\
D(\text{constant } x=n; \ d) & = \{x\} \cup D(d) \\
D(t \ \text{uz } x[e]; \ d) & = \{x\} \cup D(d)
\end{align*}
$$

Rule (ForLoop) states that the loop bounds must be public, that the variables
used in the bounds expressions cannot 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.

Rule (Call) states 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 cannot 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. Rule (Uncall) states that the type constraints for reverse
calls are the same as for forward calls.

Rule (Block) 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 5 shows the rules for how declarations extend environments.

Judgements for declarations are of the form $\rho \vdash_D d \sim \rho_1$, and state that the declaration $d$ extends the environment $\rho$ to $\rho_1$. Rule (EmptyDecl) states that an empty declaration does not change the environment. Rule (ConstDecl) extends the environment with the constant name bound to constant. Rule (VarDecl) is straightforward: The type of a variable is fetched from the environment. Rule (ArrayDecl) requires that the expression that determines the size of an array must be public, and that the array variable cannot shadow any variable used in this expression.

3.3. Procedures and programs

The rules for declarations of procedures and programs are shown in Figure 6. A judgement of the form $\vdash pgm$ states that $pgm$ is a valid program. $\vdash p \ p \sim \Gamma$ states that a procedure $p$ generates a procedure environment $\Gamma$, $\Gamma \vdash p$ states that, given the procedure environment $\Gamma$, the procedure $p$ is valid, and $\vdash A a \sim V/\tau$ states that the argument list $a$ generates the variable list $V$ and the type list $\tau$. We use $\uplus$ to append two (variable or type) lists and $\cap$ to represent the set of elements common to two lists.

Rule (Program) 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 are mutually recursive. Rule (Procedure1) builds a procedure environment for a single procedure, and rule (Procedure2) checks that a single procedure is well typed. Both use rules (ArgList), (Scalar), and (Array) for building a list of argument names and types, ensuring no name occurs twice.

3.4. Properties of the type system

The type system ensures the following properties:
\[
\begin{align*}
&\frac{}{Γ, ρ \vdash S ; \text{(Empty)}}\\
&\frac{ρ \vdash L_1 : t_0^\tau \quad ρ \vdash E : t_1 \quad R(l) \notin V(e) \cup V_I(l) \quad t_1 \sqsubseteq t_0}{Γ, ρ \vdash S^{\oplus} = e} \text{(Update)}\\
&\frac{ρ \vdash L_1 : t_0^\tau \quad ρ \vdash L_2 : t_0^\tau \quad ρ \vdash E : t_1 \quad t_1 \sqsubseteq t_0}{Γ, ρ \vdash {L_1 \leftarrow> L_2}} \text{(Swap)}\\
&\frac{R(l_1) \notin V_I(l_2) \cup V(e) \quad R(l_2) \notin V_I(l_1) \cup V(e)}{Γ, ρ \vdash E(e) \text{ if } (e) \leftarrow> l_2} \text{(SwapC)}\\
&\frac{ρ \vdash E : \text{public}}{Γ, ρ \vdash S_{s_1} ; Γ, ρ \vdash S_{s_2} ; \ V(e) \cap (U(s_1) \cup U(s_2)) = \emptyset} \text{(IfElse)}\\
&\frac{ρ \vdash E_1 : \text{public} \quad ρ \vdash E_2 : \text{public}}{(V(e_1) \cup V(e_2)) \cap U(s) = \emptyset \quad Γ, ρ[x \mapsto \text{public}] \vdash_S s} \text{(ForLoop)}\\
&\frac{Γ(f) = (τ_1, \ldots, τ_n) \quad ∀ i ∈ [1, n] : ρ \vdash L_i : τ_i \quad ∀ i, j ∈ [1, n] : i \neq j \Rightarrow R(l_i) \notin V(l_j) \cup V_I(l_i)}{Γ, ρ \vdash_S \text{call } f(l_1, \ldots, l_n)} \text{(Call)}\\
&\frac{Γ, ρ \vdash_S \text{call } f(l_1, \ldots, l_n); \quad Γ, ρ \vdash_S \text{uncall } f(l_1, \ldots, l_n);}{Γ, ρ \vdash_S \text{Call}} \text{(Uncall)}\\
&\frac{ρ \vdash D d \sim \rho_1 \quad ∀ i ∈ [1, n] : Γ, ρ_1 \vdash_S s_i}{Γ, ρ \vdash_S \{ d \ s_1 \ldots s_n \}} \text{(Block)}
\end{align*}
\]

Figure 4: Type rules for statements
\[
\begin{align*}
\vdash_D \emptyset \Rightarrow \rho &\quad \text{(EmptyDecl)} \\
\rho[x \mapsto t^2] \vdash_D d \Rightarrow \rho_1 &\quad \text{(VarDecl)} \\
\rho \vdash_D t \ u_z x \Rightarrow d \Rightarrow \rho_1 &\quad \text{(ConstDecl)} \\
\rho \vdash_E e : \text{public} \ x \notin V(e) \rho[x \mapsto t^2 \emptyset] \vdash_D d \Rightarrow \rho_1 &\quad \text{(ArrayDecl)}
\end{align*}
\]

Figure 5: Type rules for declarations

\[
\begin{align*}
\forall i \in [1,n] : \vdash_P p_i \Rightarrow [f_i \mapsto (\tau_i)] &\quad \text{(Program)} \\
\forall i,j \in [1,n] : i \neq j \Rightarrow f_i \neq f_j &
\end{align*}
\]

\[
\begin{align*}
\forall i \in [1,n] : [f_1 \mapsto (\tau_1), \ldots, f_n \mapsto (\tau_n)] \vdash_P p_i &\quad \text{(Program)} \\
\vdash_P a \Rightarrow V/\tau &\quad \text{(Procedure1)} \\
\vdash_A a \Rightarrow [x_1, \ldots, x_n]/[\tau_1, \ldots, \tau_n] &\quad \text{(Procedure2)} \\
\Gamma, [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash_S s &\quad \text{(ArgList)} \\
\vdash_A a_1 \Rightarrow V_1/\tau_1, \vdash_A a_2 \Rightarrow V_2/\tau_2 &\quad \text{(Scalar)} \\
\vdash_A a_1 \cup a_2 \Rightarrow V_1 \cup V_2/\tau_1 \cup \tau_2 &\quad \text{(Array)}
\end{align*}
\]

Figure 6: Type rules for procedures and programs
- Hermes is reversible. A proof of this is found in Appendix A.

- Secret values cannot affect public values.

- Operations on secret values are limited to a small, selected set of operations that can be implemented such that their timing do not depend on the values of their arguments. This set is parameterisable in the type system (through the $TV$ set), but must include bitwise logical operation, addition, subtraction, rotates, and comparisons (but not conditional jumps). An exception is unsafe array lookups which are not ensured to be constant time.

A proof of the latter two (excepting unsafe array lookups) this is found in Appendix B.

4. Runtime Semantics of Hermes

The operational (runtime) 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 runtime semantics also does not specify timing. It is up to implementations to ensure invariant timing of operations on secret values — except unsafe array lookups, and even those must be made as close to invariant as possible.

The runtime semantics, as specified, does not otherwise distinguish between compile time and runtime information. For example, the sizes of scalar variables and the element sizes of array are known at compile time, but in the semantics we store these sizes in the runtime environments:

**Environments** ($\eta$) bind constants to their value, variables to their integer sizes (8, 16, 32, or 64) and locations, and arrays to element sizes and locations.

**Stores** ($\sigma$) 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.

Expressions evaluate to 64-bit numbers, so when used in expressions, variable
values are extended to 64 bits. 64-bit expression values are truncated to the
size of variables or array elements when used to update these.

We use the same notation for environments as in the type semantics, but we
also use the update notation as a pattern: If $\eta_1$ is known, we use the notation
$\eta_2[x \mapsto v] = \eta_1$ to say that $\eta_2$ is equal to $\eta_1$ 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
ture 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 this property in the proof of reversibility Appendix A.

We use a family of functions $\text{newlocation}_z$ where $z$ an integer size (8, 16, 32,
or 64) that takes a store $\sigma$ and returns a new store $\sigma_1$ and a location $\lambda$ of size $z$
such that $\lambda$ is bound to zero in $\sigma_1$, and the dual function $\text{disposelocation}_z$ that
takes a store $\sigma_1$ and a location $\lambda$ and returns a store $\sigma$ obtained by removing
(unstacking) $\lambda$ from $\sigma_1$, after checking that the contents of $\lambda$ in $\sigma_1$ is 0. If not,
the result is undefined.

If $(\sigma_1, \lambda) = \text{newlocation}_z(\sigma)$, then $\sigma = \text{disposelocation}_z(\sigma_1, \lambda)$, and if $\sigma =$
dispose\_location_z(\sigma_1, \lambda) \text{ is defined, then } (\sigma_1, \lambda) = new\_location_z(\sigma) \text{.}

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

If $(\sigma_1, \lambda) = newarray_z(\sigma, vs)$, then $\sigma = disposearray_z(\sigma_1, vs, \lambda)$, and if $\sigma = disposearray_z(\sigma_1, vs, \lambda)$ is defined, then $(\sigma_1, \lambda) = newarray_z(\sigma, vs)$.

4.1. L-values and expressions

Figure 7 shows the evaluation rules for L-values and expressions. L-values evaluate to locations, and expressions to 64-bit integers. Judgements for L-values are of the form $\sigma, \eta \models_L l \in (z, \lambda)$ and state that the L-value $l$ is stored at location $\lambda$ which is of size $z$. We use a special case for constants: When $\lambda = \text{null}$, the $z$ field denotes a 64-bit constant value rather than a size. $\text{null}$ is a null location where no values are stored.

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

In the semantics for expressions, rules UnOp and BinOp use a function $Op$ that binds operator symbols to the functions they represent. So $Op(\ast)$ is a function that takes a pair of integers and returns their sum (modulo $2^{64}$) and $Op(\sim)$ is a function that takes a single 64-bit integer and returns its bit-wise negation. $Op$ is defined outside the semantic rules. Recall that comparison operators return 0 when the relation is false and $2^{64} - 1$ when the relation is true.

Rule (Variable/Constant) says that the size and location of a scalar variable and value and null location of a constant is found in the environment. Rule
\[ \sigma, \eta \vdash_\text{L} x \otimes \eta(x) \] (Variable/Constant)

\[ \eta(x) = (z, \lambda) \quad \sigma(\lambda) = (vs, ve) \quad \sigma, \eta \vdash_\text{E} e \rightarrow i \quad i < vs \] (ArrayElement)

\[ \sigma, \eta \vdash_\text{L} x[e] \otimes (z, ve[z]) \]

\[ \sigma, \eta \vdash_\text{L} \text{unsafe } x[e] \otimes (z, ve[z]) \] (UnsafeArrayElement)

\[ \sigma, \eta \vdash_\text{E} n \rightarrow n \] (Constant1)

\[ \sigma, \eta \vdash_\text{L} \otimes (n, \text{null}) \]

\[ \sigma, \eta \vdash_\text{E} l \rightarrow \sigma(\lambda) \uparrow_z \] (L-val)

\[ \eta(x) = (z, \lambda) \quad \sigma(\lambda) = (vs, ve) \] (Size)

\[ \sigma, \eta \vdash_\text{E} e \rightarrow v \] (UnOp)

\[ \sigma, \eta \vdash_\text{E} e_1 \circ e_2 \rightarrow \text{Op}(\circ)(v_1, v_2) \] (BinOp)

Figure 7: Semantic rules for L-values and expressions

:ArrayElement\) states that the location of the variable is bound in the store to a pair of vector size and vector elements, that the index expression must evaluate to a value less than the vector size, and that the location of the array element is found in the vector of elements. The type system guarantees that the location is not null and that it is bound to a pair, but it does not ensure that the index is within bounds, so this is checked at runtime. If the index it out of bounds, the effect is undefined. Rule (UnsafeArrayElement) just states that unsafe array lookups have the same runtime semantics as normal lookups.

Rule (Constant1) handles simple number constants, which evaluate to themselves. Rule (Constant2) handles named constants that are bound to pairs of values and null locations. Rule (L-val) 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. Rules (UnOp) and (BinOp) evaluate the operand(s) and then apply the semantic operator to
the value(s) of the operand(s). Finally, rule (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

Statements transform stores into stores, while keeping the environment unchanged. Judgements for running statements are of the form $\Delta, \eta \vdash s : \sigma_0 \rightleftharpoons \sigma_1$ and state that, given a procedure environment $\Delta$ and a variable environment $\eta$, a statement $s$ reversibly transforms a store $\sigma_0$ to a store $\sigma_1$. A procedure name is in a procedure environment bound to a pair of the list of the names and sizes of its parameters and its body statement.

The rules for statements are shown in Figures 9 and 10. Rule (Empty) states that empty statements do not change the store.

Rule (Update) 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 uses a function $Up$ that 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, $Up(\ll, 8)$ is a function that takes two 8-bit integers and returns the first rotated left by the second modulo 8. So $Up(\ll, 8)(0b10000001, 10) = Up(\ll, 8)(0b10000001, 2) = 0b00000110$. $Up$ is defined outside the semantic rules.

To handle $\text{uncall}$ in the semantics for statements, we need to “run” statements backwards. To this end, we use the function $I$ in Figure 8 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 $I(I(s)) = s$.

Rule (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: (CondSwap1) states that if the condition evaluates to 0 (false), there is no change in the store. (CondSwap2) 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 straightforward: If the condition evaluates to a non-zero value (rule (If1)), the first statement is executed, otherwise (rule (If2)) the second statement is executed.

Rule (ForLoop) first evaluates 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: Loop1 for when the loop counter is equal to the second bound, and Loop2 where it does not. Both use the location and the values of the bounds. Note that, to ensure reversibility, the loop counter cannot return to its initial value at the end of any iteration — if it did, we could not when running in reverse determine when to exit the loop.
\[ \Delta, \eta \models S ; : \sigma \Leftarrow \sigma \] (Empty)

\[ \sigma, \eta \models_L l @ (z, \lambda) \quad \sigma(\lambda) = v_1 \quad \sigma, \eta \models_E e \rightarrow v_2 \] (Update)

\[ \sigma, \eta \models_L l_1 @ (z, \lambda_1) \quad \sigma, \eta \models_L l_2 @ (z, \lambda_2) \]
\[ \sigma(\lambda_1) = v_1 \quad \sigma(\lambda_2) = v_2 \] (Swap)

\[ \sigma, \eta \models_E e \rightarrow v \quad v = 0 \] (CondSwap1)

\[ \sigma, \eta \models_E e \rightarrow v \quad v \neq 0 \] (CondSwap2)

\[ \sigma, \eta \models_E e \rightarrow v \quad v = 0 \] (If1)

\[ \sigma, \eta \models_E e \rightarrow v \quad v \neq 0 \] (If2)

\[ (\sigma_1, \lambda) = \text{newlocation}_{64}(\sigma) \quad \sigma_2 = \sigma_1[\lambda := v_1] \] (ForLoop)

\[ \sigma(\lambda) = v_2 \] (Loop1)

\[ \sigma(\lambda) \neq v_2 \] (Loop2)

Figure 9: Semantic rules for statements (part 1)
∀i ∈ [1, n]: σ, η |= L lᵢ @ (zᵢ, λᵢ) \[\Delta(f) = \{([x₁, z₁], \ldots, [xₙ, zₙ]), s]\]

∀i ∈ [1, n]: σ, η |= L lᵢ @ (zᵢ, λᵢ) \[\Delta(f) = \{([x₁, z₁], \ldots, [xₙ, zₙ]), s\} \models_S s : σ ⇔ σ₁\]

(Call)

\[\Delta, η \models_S \text{call } f(l₁, \ldots, lₙ); : σ ⇔ σ₁\]

(Uncall)

\[\eta, σ \models_D d \sim η', σ₀\]

\[\forall i ∈ [1, n]: \Delta, η' \models_S sᵢ : σᵢ₋₁ ⇔ σᵢ\]

\[\eta', σᵣ \models_D^{\text{inv}} d \sim η, σᵣ₊₁\]

\[\Delta, η \models_S \{d s₁ \ldots sₙ\} : σ₀ \sim σᵣ₊₁\]

(Block)

Figure 10: Semantic rules for statements (part 2)

Rule (Call) finds the sized locations of the arguments, looks the procedure up in the procedure environment to get the list of parameter names and the body of the procedure. It then creates a new environment that binds the parameter names to the argument locations and executes the body in this environment.

Note that no new locations are allocated — parameters are passed by reference, so the local parameter identifiers are bound to the locations that are passed in.

Rule (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.

Rule (Block) uses the declarations to extend the environment and store, executes the body, and uses the declarations to restrict the store.

4.3. Declarations

The rules for declarations is shown in Figure 11. There are two kinds of judgements for declarations: η₀, σ₀ \models_D d \sim η₁, σ₁ says that the declaration d extends η₀ and σ₀ to η₁ and σ₁. Dually, η₀, σ₀ \models_D^{\text{inv}} d \sim η₁, σ₁ says that “undoing” the declaration d restricts η₀ to η₁ and σ₀ to σ₁.

Rules EmptyDecl and EmptyDeclInv say that the empty declaration has no
effect in either direction. Rules ConstDecl (ConstDeclInv) state that a constant declaration extends (retracts) the environment but leaves the store unchanged. Recall that constants are stored in the environment by using a null location. Rules for variable and array declarations do not distinguish secret and public values (that is done by the type system). In the forwards direction (rule (VarDecl)), a new location (bound to zero) is created for the variable and the variable is bound to the location. In the backwards direction (rule (VarDeclInv)), disposelocationz verifies that the location is bound to zero before it is removed from the store. In the forwards direction (rule (ArrayDecl)), a new ze-
roed array is created in the store and the variable is bound to its location in the
environment. In the backwards direction (rule (ArrayDeclInv)), \texttt{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, so
allocations are matched by deallocations in a LIFO manner, preserving stack
discipline.

4.4. Procedures and programs

The rules for procedures and programs are shown in Figure \ref{fig:procedures}. There is no
\texttt{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 (rule
(Program)) is just creating a procedure environment \(\Delta\). 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. Rule (Procedure) creates a
procedure environment (using rules (ArgList), (Scalar), and (Array)) for a single
procedure, which 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
rule (Program). Rules (Xcall) and (Xuncall) describe external calls to Hermes.
These are very like rules (Call) and (Uncall) 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 show and discuss a number of code examples that compare C and Hermes,
but first we clarify some details.

The core language syntax in Figure \ref{fig:syntax} does not specify operator precedence
or parentheses. We will use parentheses and the following operator precedences
in the code examples discussed below.
∀i ∈ [1, n]: ⊨ p_i ≫ Δ_i (Program)

⊨ A a ↣ x s (Procedure)

⊨ A a_1 ↣ x s_1 , A a_2 ↣ x s_2 (ArgList)

⊨ A t u z x ↣ [(x, z)] (Scalar)

⊨ A t u z x [.] ↣ [(x, z)] (Array)

\[ \Delta(f) = ([x_1, z_1], \ldots, (x_n, z_n), s) \]

\[ \begin{array}{c}
\Delta, [x_1 \mapsto (z_1, \lambda_1), \ldots, x_n \mapsto (z_n, \lambda_n)] \models S(s) : \sigma \vdash \sigma_1 \\
\Delta, \sigma, [\lambda_1, \ldots, \lambda_n] \models X \text{ call } f \Rightarrow \sigma_1
\end{array} \] (Xcall)

\[ \begin{array}{c}
\Delta(f) = ([x_1, z_1], \ldots, (x_n, z_n), s) \\
\Delta, [x_1 \mapsto (z_1, \lambda_1), \ldots, x_n \mapsto (z_n, \lambda_n)] \models I(s) : \sigma \vdash \sigma_1
\end{array} \] (Xuncall)

Figure 12: Semantic rules for procedures and programs
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 $l \oplus= (e_1 \oplus= 0) \& (e_2)$; 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 $\oplus= 0$ 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 $s_1 \, s_2 \, I(s_1)$, where $I(s_1)$ is the inverse of $s_1$, so for brevity and clarity we have added a shorthand for this: $s_1 \, @ \, s_1$. This is expanded into the full form in the parser. @ is right-associative, so $s_1 \, @ \, s_2 \, @ \, s_3$ is equivalent to $s_1 \, @ \{ s_2 \, @ \, s_3 \}$ and expands to $s_1 \, s_2 \, s_3 \, I(s_2) \, I(s_1)$.

@ binds more strongly than statement sequence, so $s_1 \, @ \, s_2 \, s_3$ is equivalent to

{$s_1 \, @ \, s_2$} $s_3$ and expands to $s_1 \, s_2 \, I(s_1) \, s_3$.

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

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 */
}

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

Figure 13 (bottom) shows Hermes code for the TEA encryption algorithm [23], a simple cipher used mainly for teaching. Only the encryption function is shown — decryption is done by uncalling the encryption function. The sizes of v and k are 2 and 4, respectively. Compare to the equivalent program in C (taken from the Wikipedia article for TEA [24]) at the top of Figure 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 this requires that the C compiler should not remove assignments to dead variables. When we move parameters to and from local variables, we use <-> when we want to replace the original value with a modified value (such as replacing clear text with cipher text) and += and -= when the local variable doesn’t change (such as when having a local copy of a key). We could use <-> for both, but this is more expensive, so we use += and -= when we can.

5.2. RC5

Figure 14 shows Hermes and C code for the central part of RC5 [1], another simple cipher. The Hermes program shows size a 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 (taken from the Wikipedia article for RC5 [25]) uses a macro for this (which most C compilers can optimise to a rotate). 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) expands a 128-bit key to 800 bits in a non-reversible (and relatively expensive) process. Due to the cost, key expansion is typically done once before a number of blocks are encrypted using the same key. In Hermes, an additional uncomputation of


```c
#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;
}
```

```c
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];
        i++;
      }
    }
}
```

Figure 14: RC5 core in C (top) and Hermes (bottom)

the expanded key is required after encrypting the blocks, so the expanded key is cleared. Since key expansion is not reversible, Hermes needs to store intermediate values in a “garbage” array. The garbage array is reset to zeroes when the expanded key is uncomputed by uncalling the key expansion procedure.

5.3. Speck128

Figure 15 shows code in C (top) and Hermes (bottom) for speck128 [2] (a cipher used by NSA). The C version is taken from the Wikipedia article for Speck [26]. 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
```c
#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) {
            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 15: Speck128 in C (top) and Hermes (bottom)
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(2⁸). These are used in a matrix multiplication in the field GF(2⁸) 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 lookups, as they represent multiplication with the neutral element of multiplication. For details about multiplication in the field GF(2⁸), we refer the reader to the AES Standard document [19], but the discussion below does not require understanding the theory.

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:

```c
for (i = 0; i < 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

```c
{ u8 x1;
  x1 += S[x];
}
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 lookups 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 lookups must be used for S-box lookups. Unsafe lookups imply that the array itself must be declared to have secret content, so both \( S \) and \( S_{\text{inv}} \) must be declared as secret and the two code snippets above must be modified to use unsafe lookups whenever secret data is used as index:

```c
for (i = 0; i < 256) {
    unsafe S_{\text{inv}}[S[i]] += i;
    i++;
}
```

```c
{ u8 x1;
    x1 += unsafe S[x];
    x -= unsafe S_{\text{inv}}[x1];
    x <=> x1;
}
```

The unsafe lookups allow AES and other ciphers using S-boxes to be implemented in Hermes, but there is no guarantee of constant timing. Bernstein [20] 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.

It is possible to implement AES without using unsafe lookups in S-boxes: You can compute \( S[i] \) from scratch every time instead of caching it in a table. Entries in the Rijndael S-box used in AES can be calculated using logical operations, as can any reversible function on bounded integers. This method is often
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 16: C code for AES
called *bit-slicing* because you can compute the function for N values in parallel using N-bit operations. The number of required logical operations can be very large (more than 100 for the AES S-box [27]), so even when exploiting bit parallelism, it can be much more expensive than table lookups. Implementing the mix-columns step using reversible logic operations is fairly fast, but the inverse (required for decryption) is very time-consuming to do without using tables.

The C code for the AES cipher shown in Figure 16 is based on AES Dust [28], but modified slightly (which does not negatively affect performance). The $M_{0x3}$ 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 17 and 18. The part shown in Figure 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 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 increases the execution time, it allows us to use the same function in reverse for decryption, and it ensures that secret values are cleared.

One thing to note about AES is that decryption is not a straightforward inverse of encryption. It relies on multiplication in the field GF($2^8$) of a matrix that is the inverse of the one used for encryption. Hermes requires that decryption is a simple inversion of decryption, so implementing AES in Hermes requires both matrix multiplications to be done both when encrypting and decrypting, which is a significant overhead.
```c
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; i< 16) { k[i] = mk[i]; i++; } }
    // copy 28-bit master key to x
    for(i=0; i< 16) { x[i] = s[i]; i++; }
    for(round = 0; round < 11) { // AddRoundKey
        for(i=0; i< 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; i< 4) { // ExpandKey
                t = k[12+((i-3)&3)]; @ k[i] = unsafe sbox[t];
                i++;
            }
            for(i=0; i< 12) { t = k[i]; @ k[i+4] = t;
                i++;
            }
            for (i=0; i< 16) { // SubBytes and ShiftRows
                x[((13*i)&15] = unsafe sbox[s[i]];
                s[i] = unsafe sboxInv[x[((13*i)&15]];
                i++;
            }
        } else; // if not last round
        if(round!=9) { // MixColumns
            for(i=0; i< 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 M0xe[b*c] = a*d;
                x[i+2] = unsafe M0xb[c*d] = a*b;
                x[i+3] = unsafe M0xd[a*d] = b*c;
                a = unsafe M0xe[x[i]] = unsafe M0xb[x[i+1]]
                = unsafe M0x9[x[i+2]] = unsafe M0xe[x[i+3]];
                b = unsafe M0xe[x[i]] = unsafe M0xb[x[i+1]]
                = unsafe M0xe[x[i+2]] = unsafe M0x9[x[i+3]];
                c = unsafe M0xe[x[i]] = unsafe M0xb[x[i+1]]
                = unsafe M0x9[x[i+2]] = unsafe M0xe[x[i+3]];
                d = unsafe M0xe[x[i]] = unsafe M0xb[x[i+1]]
                = unsafe M0x9[x[i+2]] = unsafe M0xe[x[i+3]];
                i += 4;
            }
        } else; // else
        round++;
    }
}
```

Figure 17: Hermes code for AES (part 1)
\begin{figure}
\begin{verbatim}
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 {
    }
}
\end{verbatim}
\caption{Hermes code for AES (part 2)}
\end{figure}

6. Optimising Hermes with Partial Evaluation

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

- bound check at array lookup.
- 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.

These checks, unless removed by optimisation, make Hermes slower than the equivalent C program. Array bound checks can often be eliminated using standard data-flow techniques \cite{29}, but the other checks listed above can be difficult to remove by simple static analysis.

One option is to assume that the assertions hold and ignore them. This is essentially the approach taken by C. This is unsafe, but with thorough testing the probability that the checks will never fail can be made very low. Another
possibility is to exploit the fact that array sizes and indexes (except for unsafe lookups) 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 \cite{12, 13} 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 bound 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 runtime checks that are not removed are bound checks for unsafe array lookups and checks that secret local variables are zeroed at the end of their scope.

As computations on public data are eliminated, the residual program will do fewer computations than the original program. Furthermore, since there is no conditional branching, side-channel attacks that target weaknesses in speculative execution (such as Spectre \cite{30}) are avoided.

Partial evaluation has already been implemented for the reversible language Janus \cite{14, 31}. A complication in that work was that not all control could be eliminated by partial evaluation, as dynamic conditions for control structures were allowed, where we in Hermes only allow static (public) conditions. Furthermore, the control flow present in residual Janus programs could not always be expressed using the structured control structures of Janus, so unstructured control was added. By restricting conditional control flow to use static conditions, we can in the partial evaluator for Hermes eliminate all control. 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 `disposearrayz`.

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.

Since program control is entirely public, complete unrolling of loops and inlining
of procedure calls is safe: If this doesn’t terminate, running the original program
with the same public data will not terminate either.

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
runtime checks.

As an example, reifying the checks in the Speck128 procedure in Figure 15
results in the program shown in Figure 19 (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 20.

Note that the only residual assertions are that the local variables are zeroed at
the end of their scope.
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; i<31) {
        call Rp(a, b, i); i++;
        call Rs(x, y, b); assert i != 0;
    }
    for (i=31; i>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 19: Speck128 with reified checks (abbreviated)
```c
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;
            a >>= 8; a += b; a ^= 28; 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 */
        }
        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 20: Residual Speck128 program (somewhat abbreviated)
7. Compilation and Benchmarks

We implemented a prototype compiler for the subset of Hermes output by the
partial evaluator, targeting x86-64 through 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 object code size and tim-
ings 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 run-
ing 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 in-
line 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 Red Pike.

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 Red Pike 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 not only key ex-
expansion but also the mix-columns step by multiplying with the inverse matrix,
which adds a significant overhead. AES has nested loops with bodies of signifi-
<table>
<thead>
<tr>
<th>Program</th>
<th>object code size</th>
<th>time</th>
<th>Hermes/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEA (C)</td>
<td>1464</td>
<td>11.29</td>
<td></td>
</tr>
<tr>
<td>TEA (Hermes)</td>
<td>4440</td>
<td>11.30</td>
<td>1.00</td>
</tr>
<tr>
<td>Speck 128 (C)</td>
<td>1432</td>
<td>5.30</td>
<td></td>
</tr>
<tr>
<td>Speck 128 (Hermes)</td>
<td>3304</td>
<td>8.83</td>
<td>1.67</td>
</tr>
<tr>
<td>RC5 (C)</td>
<td>1400</td>
<td>2.56</td>
<td></td>
</tr>
<tr>
<td>RC5 (Hermes)</td>
<td>1888</td>
<td>2.98</td>
<td>1.16</td>
</tr>
<tr>
<td>Red Pike (C)</td>
<td>1400</td>
<td>3.14</td>
<td></td>
</tr>
<tr>
<td>Red Pike (Hermes)</td>
<td>2160</td>
<td>3.51</td>
<td>1.12</td>
</tr>
<tr>
<td>AES (C)</td>
<td>2176</td>
<td>4.26</td>
<td></td>
</tr>
<tr>
<td>AES (Hermes)</td>
<td>58824</td>
<td>13.85</td>
<td>3.25</td>
</tr>
</tbody>
</table>

Table 1: Comparison of C and Hermes
cant size, so the loop unrolling 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.

A student project has ported the compiler to generate ARM-64 code which was tested using a Raspberry Pi computer. In this setting, the code generated by the Hermes compiler was slower than those generated by GCC even for the cases where there was little difference above (such as TEA, RC5 and Red Pike). By comparing the output of the two compilers, the students observed that GCC did significantly more reordering of instructions, so we suspect compile-time instruction scheduling is required to get closer to GCC performance on ARM.

8. Runtime Errors

Some important properties of Hermes: Reversibility and non-leakage depend on programs running to completion without errors, and the operational semantics does not specify what happens if an assertion in a semantic rule fails.

If a program stops with an error message, this fact can in itself leak some information if, for example, the error is that a secret variable is not cleared to zero at the end of its scope. And if the memory is not cleared when a program stops with an error, the memory contents can leak additional information. And if the memory is cleared, it is not possible to run the program backwards from the erroneous state.

When 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 control flow cannot depend on secret values, partial evaluation unrolls the control flow completely. Hence, the possible run-time errors in the resulting program are relatively few: A secret variable or array is not cleared at the end of its scope, an unsafe array lookup is outside the bounds of the array. Division by zero can, for example, not occur, as division is only allowed on public values, and
unbounded recursion would be revealed during partial evaluation.

Runtime error messages can be helpful in locating errors, so we suggest that these are reported at least during testing, but when using Hermes in secure applications, it may be better to continue execution with some default behaviour that will not reveal secrets.

For example, instead of checking that a secret variable or array is cleared at the end of its scope, it can be forcibly cleared. This will not affect timing, and it will not leak information through uncleared memory. It can, however, mean that an encrypted message cannot be decrypted again, as some information is lost when a nonzero variable is cleared.

When doing an unsafe array lookup, this can be made modulo the array size. This requires dividing a secret index by a public array size, so this can potentially leak information through timing differences in the division operation. But unsafe array lookups are not guaranteed to be time invariant anyway, so this is not a major issue. Besides, S-boxes tend to use powers of two as sizes, so the index can be restricted to the correct range by a bitwise AND. Information is lost when the index is modified, so reversibility is also here compromised.

In conclusion, it is possible to avoid run-time errors causing timing leaks, but the resulting encrypting messages can be corrupted.

So we suggest testing Hermes programs using an execution mechanism that reports errors (such as the Hermes interpreter). Since all control flow is determined by public input, which does not change, even a little testing will usually reveal errors, and if no errors are found, the above methods can be used to reduce the impact of undiscovered errors.

9. 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 at-
tacks and leaks to memory.

While this paper only 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. As 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 is 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 runtime behaviour. These semantics has been used to argue both that secret information does not leak into public variables and that type-correct programs are, indeed, reversible. In appendices to this paper, we have argued reversibility and safety for programs that terminate without error.

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 runtime 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 runtime 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, by unrolling loops and inlining procedures, increases the code size, often dramatically. Since its main purpose is to remove unnecessary runtime 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 runtime tests. This will, however, complicate the subsequent compilation, and not avoid attacks that exploit weaknesses in speculative execution.

We also have prototype compilers for earlier versions of Hermes to C and WebAssembly [11]. These cannot 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 implemented 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 lookups to the current version. Unsafe lookups 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 in part because Hermes does array bounds check when the array index is secret (i.e., for unsafe array lookups) and in part because Hermes needs to do twice as many matrix multiplications, as explained in Section 5.4. Using bit-slicing instead of unsafe array lookups (see Section 5.4) can avoid array bounds checks (at the cost of adding a large number of bitwise logical operations), but it does not reduce the number of matrix multiplications, which is doubles in Hermes to make decryption a simple inverse of encryption (see Section 5.4). This suggests that Hermes is best suited for ciphers where decryption is a simple inverse of encryption, which is the case for all the ciphers discussed in this paper with the exception of AES.

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).

At the moment, Hermes itself has no feature for declassification [21]: Casting secret values to public. This has to be done outside Hermes, e.g. when a C program that calls Hermes sends an encoded message through a non-secure channel. Declassification is trivial to add to the Hermes type system – it is just a cast from secret to public – but it will no longer be true that all public values can be computed in the partial evaluation stage – declassified secret values are public but not known. A solution could be to introduce a third type: declassified, where declassified values can be used where public values can be used, but are not evaluated by the partial evaluator. Having a declassified type would also help keeping track of where declassified values flow. Secret values can be explicitly converted to declassified and vice versa, but neither can be converted to truly public.

Sometimes it can be convenient to treat a 64-bit value as eight 8-bit values,
and currently there is no easy way to do this in Hermes. So adding something like FORTRAN equivalence or common declarations that allows an array to be viewed as both an array of 64-bit values and an array of 8-bit values could be useful. As an example, the Hermes AES implementation currently does everything on 8-bit values, but could benefit from doing some operations using 64-bit values.

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.

Acknowledgements. 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. Thanks in particular to students Jonas Grønborg and Christian Marslev, who ported the current compiler to ARM.

References


URL https://doi.org/10.1007/978-3-319-99498-7\_23


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

URL http://drops.dagstuhl.de/opus/volltexte/2018/8477


URL http://doi.acm.org/10.1145/3140587.3062363


URL http://dblp.uni-trier.de/db/conf/pepm/pe1998.html


URL https://doi.org/10.1145/325694.325702

URL https://doi.org/10.1145/1866307.1866341


URL https://pdfs.semanticscholar.org/fbf1/8eb234f19897f758fedebe9c52f29fbbf15a.pdf


Appendix A. Proof of Reversibility

While Janus, from which Hermes is inspired, has been proven reversible [6], Hermes is sufficiently different from Janus that this proof does not carry over. In particular, the loop construct in Hermes is very different from the one used in Janus. So we make a new proof.

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 8. More precisely, we will prove that if \( \Delta, \eta \models \models_S s : \sigma_1 \rightleftarrows \sigma_2 \), then \( \Delta, \eta \models \models_S I(s) : \sigma_2 \rightleftarrows \sigma_1 \). 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 \( \Delta, \eta \models \models_S \; : \sigma \leftarrow \sigma \), 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 \( l \oplus=1 \; e; \; l \oplus=2 \; e; \), \( e \) evaluates to the same in both updates. Also, \( l \) evaluates to the same sized location \((z, \lambda)\) both times, as any index expression used in \( l \) is unaffected by the updates. So, if the value stored at \( \lambda \) before the sequence is \( v_1 \) and \( e \) evaluates to \( v_2 \), the net effect is to update \( \lambda \) 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 \( \ast=, \; \oplus=2 \) is also \( \ast=, \) so \( U(\oplus=2 \; (U(\oplus=1 \; (v_1, v_2 \downarrow z)), v_2 \downarrow z)) \) reduces to \( (v_1 \ast (v_2 \downarrow z)) \ast (v_2 \downarrow z) \), where \( \ast \) is bit-wise exclusive OR. By associativity, we get \( v_1 \ast ((v_2 \downarrow z) \ast (v_2 \downarrow z)) \), which is equal to \( v_1 \ast 0 \), which is equal to \( v_1 \), which was the original value of \( \lambda \).

If \( \oplus=1 \) is \( \ast\neq, \; \oplus=2 \) is \( \ast\neq, \) so we by similar reasoning get \( (v_1 \ast (v_2 \downarrow z)) - (v_2 \downarrow z) \), which is equal to \( v_1 \) (since we operate modulo \( 2^z \)). The case for \( \ast\neq \) is similar.

**Swap** A swap statement reverses to itself.

The restrictions on swap statements ensure that the two locations \( \lambda_1 \) and \( \lambda_2 \) are different, and that neither is used when calculating the L-values. So in the sequence \( l_1<->l_2; \; l_1<->l_2; \), 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 \( \lambda_1 \) and \( \lambda_2 \) uses the contents of \( \lambda_1 \) and \( \lambda_2 \), so updating \( \lambda_1 \) and \( \lambda_2 \) 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 (CondSwap1)), 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 (Loop1), the store is unchanged by the loop body. The only other changes to the store are creating a location $\lambda$ for the loop counter $x$, giving $\lambda$ the value of the initial (and final) loop bound, resetting $\lambda$ to zero, and then disposing of it. By the informal definitions of *newlocation* and *disposelocation*, the net effect of this is no effect.

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

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

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 (Loop2)) different from both $v_1$ and $v_2$, which we have asserted in comments. If $x$ is equal to $v_2$ between iterations, the loop would terminate earlier. If $x$ is equal to $v_1$ between iterations, backwards execution would terminate earlier.

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 cannot 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

```c
{ 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 envi-
ronments 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 \( \eta, \sigma \models_D d \leadsto \eta', \sigma_0 \), then \( \sigma_0(\lambda) = \sigma(\lambda) \) for all locations \( \lambda \) in \( \eta \), and \( \sigma_0 \) binds all \( \lambda \) that are in \( \eta' \) and not in \( \eta \) to either 0 or arrays with all elements bound to 0.

2. If \( \eta', \sigma_0 \models_D^{inv} d \leadsto \eta, \sigma_{n+1} \), then \( \sigma_n(\lambda) = \sigma_{n+1}(\lambda) \) for all locations \( \lambda \) in \( \eta \), and \( \sigma_n \) binds all \( \lambda \) that are in \( \eta_n \) and not in \( \eta \) 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 \( \eta \)) 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 \( \eta_{n+1} \), 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 them 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. See 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.

### Appendix B. Proof of Safety

We prove noninterference in the sense used by Hedin and Sabelfeld [21], but restricted to termination-sensitive noninterference: If a program stops without error, no secret information is leaked. Hedin and Sabelfeld define noninterference as indistinguishability of two executions with equivalent public (low) input but different secret (high) input in the sense that execution time and public output will be the same. We prove this by arguing that secret values will neither affect the values of public variables nor affect execution time. This implies that, with identical public input, different secret inputs will yield the same public output and the same execution time.

We omit unsafe array lookups from the proof (as these can, indeed, leak information), and we assume programs terminate without error. See Section 8 for a discussion about how to avoid leaks from runtime errors.

Note that the proof only states that the operations can be implemented safely. It is up to a specific compiler to ensure this. Since we need only prove the existence of a compiler, we can assume that the partial evaluator is used as a prepass to code generation. Equivalent public inputs yield identical residual programs without public parameters, so they have no public output. So the part of the noninterference property that concerns public output is trivially true. We
just need to prove that execution time does not depend on secret values.

By partial evaluation, public variables and updates are eliminated, public expressions are replaced by constants, procedure calls and named constants are inlined, all conditionals replaced by the chosen branches, and all loops unrolled. We will also replace conditional swaps by unconditional code as described in Section 3.2. This reduces the language to a subset described in Figure B.21. Note, also, that we have added assert statements, which after partial evaluation are either testing if a variable is zero or an array of constant size is all zero. Since we assume programs terminate without error, we can assume all assertions succeed.
As all control structures are eliminated, programs are non-branching, so the same sequence of instructions is executed every time the programs run. Different executions may have different subsets of these instructions in cache, so the actual runtime may differ. Similarly, the arrays passed to the Hermes program may be more or less cached, which also affects running time. It is obviously possible for a program that calls the Hermes program to store secret data in such a way that the cache behaviour can leak information about the values (for example by ensuring everything is cached if the key is odd, and nothing is cached if the key is even). So we must necessarily make the assumption that the caching pattern of instructions and data prior to calling Hermes does not in itself reveal secret information, i.e., that whether or not an instruction or value is cached can depend on the location of the instruction or value, but not be influenced in any way by the values of secret data. So we will not compare running times of two different executions with different secret data (as we can not ensure that these are identical), but instead argue that the running time of a single execution does not depend on secret data if the cache pattern prior to execution does not.

The proof refers to the syntax in Figure B.21 and to the semantic rules in Section 4.

Calling Hermes

Calling Hermes from other programs is covered by rule (Xcall) in Figure 12. Given a store $\sigma$ and a list of locations $\lambda_i$ for the parameters, the rule builds an environment and executes the procedure body $s$ to produce a new store $\sigma_1$. Note that the environment only depends on the location of the parameters and not on their values (which are stored in $\sigma$). So it is only the execution of $s$ that can leak information about secret values. In compiled code, the environment is actually eliminated, as parameters are represented (following the procedure call standard of the target machine) by registers or stack locations that at call time contain the locations of the parameter values. This is consistent with the conclusion that only execution of $s$ can reveal secret information.
Executing statements

Semantics for execution of statements is covered in Figures 9 and 10. Only some of the rules are relevant for the subset we consider: (Empty), (Update), (Swap), and (Block). Additionally, we must consider assert statements. As we assume that programs terminate without error, executing an assert statement consists of evaluating the condition and verifying its result. No jump is made.

(Empty) No code is executed, and the store is unchanged, so no information is leaked.

(Update) The L-value is resolved to a location $\lambda$, the contents $v_1$ of this is fetched, the expression is evaluated to a value $v_2$, the update operator $\oplus$ is applied to the two values, and the result is stored at $\lambda$ in the store. We note that we assume that timing of memory accesses only depend on the locations and not the values that are loaded and stored, so fetching from and storing to $\lambda$ will not reveal secrets through timing. The contents of $\lambda$ are secret (as we have eliminated all public variables and arrays), so storing into this will not leak secret information to non-secret variables. Assuming the update operator $\oplus$ can be implemented in constant time, only resolving the L-value and evaluating the expression can potentially leak information. We will look at L-values and expressions in a bit.

(Swap) The two L-values are resolved to two locations $\lambda_1$ and $\lambda_2$, values $v_1$ and $v_2$ are fetched from these, and these are stored in $\lambda_2$ and $\lambda_1$, respectively. By the assumption that memory accesses do not leak information about the values that are transferred, it is only resolving L-values that can potentially leak information.

(Block) The declarations extend the environment and the store, the statements are executed in sequence, producing new stores, and the final store is modified by declarations. By induction, the statements in the body of the block do not leak information, so it is only handling declarations that can potentially do this. We look into this in a bit.
**Assertions** Checking if a variable is zero consists of finding its location, fetching the value at this location, and verifying that it is zero. Finding a location of a variable is considered under resolving L-values below, and fetching the contents of a location is assumed to not reveal information about the fetched value. Verifying that it is zero can be done by a conditional branch. Since we assume assertions succeed, the branch is never taken, so no information is leaked. Checking if an array is all zero can be done by ORing all elements and then checking if the result is zero. Finding the location of the elements is considered under resolving L-values below, and fetching the elements is assumed not to reveal information. ORing can be done in constant time, and the check is again done by a non-taken conditional branch. So no information is leaked – under the assumption that programs terminate without error.

**Resolving L-values**

Semantics for L-values are presented in Figure[7] Only rules (Variable/Constant) and (ArrayElement) are relevant.

(Variable/Constant) Named constants are eliminated by partial evaluation, so only variables are relevant. The location of the variable $x$ is found by looking up in $\eta$. This does not depend on any secret values. When a program is compiled, the lookup is done at compile time and binds $x$ to either a register or a stack offset. This is consistent with not leaking information.

(ArrayElement) The location $\lambda$ of the array is found (at compile time) by looking $x$ up in $\eta$. This location is accessed to find the array size $vs$ and the vector of elements $ve$. After partial evaluation, the array size $vs$ and the index $i$ are known constants, so in the compiled code, the test is eliminated and $\lambda$ will only contain a reference to $ve$. So the only generated code is adding a constant ($i$ scaled by the element size) to the location of $ve$. This can not leak secret information.
Evaluating expressions

Semantics for expressions are presented in Figure 7. Only rules (Constant1), (L-val), (UnOp) and (BinOp) are relevant.

(Constant1) No evaluation is done, so nothing is leaked.

(L-val) As named constants are eliminated by partial evaluation, the location \( \lambda \) cannot be null. Fetching the contents of \( \lambda \) in the store \( \sigma \) can by the assumption on memory accesses not reveal secret information.

(UnOp) By induction, evaluation of the subexpression \( e \) does not leak information. The only unary operation is binary negation, which can be implemented by a single constant-time instruction, so it does not leak secret information.

(BinOp) By induction, evaluation of the subexpressions \( e_1 \) and \( e_2 \) does not leak information. Since the values \( v_1 \) and \( v_2 \) are secret, the set of binary operations \( \odot \) excludes operations in the set \( TV \) of time-variant operations. So applying \( \odot \) to \( v_1 \) and \( v_2 \) can be implemented in constant time.

Handling declarations

Semantics for declarations are presented in Figure 11. Only rules (EmptyDecl), (EmptyDeclInv), (VarDecl), (VarDeclInv), (ArrayDecl) and (ArrayDeclInv) are relevant.

(EmptyDecl) Nothing happens, so no information is leaked.

(EmptyDeclInv) Nothing happens, so no information is leaked.

(VarDecl) newlocation \( z \) is applied to the store \( \sigma \) to create a location \( \lambda \) for the variable \( x \) and update the store to map this location to 0. In compiled code \( \lambda \) is either a register or the top of the stack, which has been extended to hold this. In either case, storing a zero in this location can be done in
constant time and certainly does not leak secrets in any other way. By
induction, the remaining declarations \( d \) do not reveal information.

(VarDeclInv) The location \( \lambda \) of the variable is found in \( \eta \) (at compile time),
and \( \text{disposelocation}_z \) is applied to \( \lambda \) and the store \( \sigma_1 \) to check that the
location contains zero and to remove it from the store. The zero-check
has been extracted as an assertion, so only the removing of the location
is done in the compiled code for (VarDeclInv). This takes no time if the
location is a register and constant time if the location is the stack top. By
induction, the remaining declarations \( d \) do not reveal information.

(ArrayDecl) \( \text{newarray}_z \) is called on the store \( \sigma \) and a constant array of size \( n \).
This allocates space for the array on the stack and initialises the elements
to 0. This does not depend on any secret values and can be done in
constant time. By induction, the remaining declarations \( d \) do not reveal
information.

(ArrayDeclInv) The location \( \lambda \) of the array is found in \( \eta \) (at compile time),
and \( \text{disposearray}_z \) is applied to \( \lambda \) and the store \( \sigma_1 \) to check that the
array is all zero and to remove it from the store. The zero-check has been
extracted as an assertion, so only the removing of the location is done in
the compiled code for (ArrayDeclInv). This is done by moving the stack
pointer, which is constant time. By induction, the remaining declarations
\( d \) do not reveal information.

This completes the proof that partially evaluated Hermes programs can be im-
plemented without leaking information about secret values through timing or
dataflow to public variables (of which there are none). This does not imply that
partial evaluation must be done to avoid leaks, it just shows one possible way
to implement Hermes without leaks through timing or data flow.