Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C

EADS IW/SE Technical Note 2008-IW-SE-00010-1

Charles Hymans[1] and Olivier Levillain[2] *

[1] EADS Innovation Works IW/SE/CS, 12, rue Pasteur - 92152 Suresnes - France, charles.hymans@eads.net

[2] Direction Centrale de la Sécurité des Systèmes d'Information SGDN/DCSSI/SDS, 51, bld la Tour Maubourg - 75007 Paris - France, olivier.levillain@sgdn.gouv.fr

Abstract. Static analysis is a difficult task, partly because programming languages are extremely rich, and have intricate semantics with architecture-dependent aspects. We have therefore chosen to design Newspeak, a kernel language dedicated to the purpose of static analysis. And, we have implemented a front-end, C2Newspeak, that translates C programs into Newspeak. Thus, any static analysis algorithm that uses this front-end, is preserved from the aforementioned sources of complexity. This paper fully presents the syntax and precise semantics of Newspeak. The design rationale of the language is explained and the advantages for static analysis highlighted. The various details of the translation from C to Newspeak are shown on examples. C2Newspeak was made to compile embedded C programs of a few million lines of code. It is, as well as a few other utilities, provided as free software under the LGPL.

1 Introduction

Following a trend found in most other industries, increasingly more functionalities on aeronautical products are implemented in software. As a result, software grows both in size and complexity, making the task of verification harder. At EADS, we build static analysis tools to automatically ensure programs have their expected behavior. We focus particularly on C, which is one of the most common programming language found on embedded software.

The C programming language's syntax and semantics are extremely complex. Many constructions are redundant, some semantics information may be missing, or depend on the code context, the compiler or the hardware. It is therefore not well-advised to perform static analysis directly on the syntax tree of a C program. This approach produces a tangled, unnecessarily large, hard to maintain, and laden with corner bugs implementation. Thus annihilating any trust one could put in the results of the tool.

So we decided to create an intermediate language, more practical than C for static analysis uses. The goal was to obtain a language that would be:

* This work was realized during an internship at EADS Innovation Works.

- precise: its semantics is precisely defined,
- simple: primitives are as few, as classical and as concise as possible,
- minimal: no language primitive or fragment of a primitive should be expressible as a combination of other primitives,
- explicit: primitives are context-free, i.e. all semantic information needed to execute any primitive are readily available,
- analysis-oriented: the language primitives have annotations useless to the execution but necessary for a static analysis to perform correctness checks,
- architecture-independent: all architecture dependent features (essentially the size of types and the offsets of structure fields) are already computed and made explicit during the translation to Newspeak,
- expressive: it should be possible to translate all C in Newspeak.

This language is called Newspeak in reference to George Orwell's novel "*1984*":

> In the end the whole notion of goodness and badness will be covered by only six words – in reality, only one word. Don't you see the beauty of that, Winston?

The compilation from C to Newspeak is implemented as a front-end called C2newspeak. This translation is faithful to the semantics of C as described in the ANSI standard [7]. It is parameterized by various architecture and compiler (integer widths, padding...) specifics. C2newspeak was made to scale up to a few million lines of code. It is distributed with other utilities under the LGPL at `http://www.penjili.org/newspeak.html`.

The syntax and semantics of Newspeak are presented in section 2. The compilation is explained through examples in section 3. Implementation, experiments and some other Newspeak utilities are all discussed in section 4. We compare other intermediate languages in section 5 on related work. At last, possible further extensions are listed before concluding in section 6.

2 Syntax and Semantics

2.1 Newspeak essential paradigms

In order to concisely expose the essential paradigms of Newspeak, let us start with the succinct subset of figure 1.

For now, the only type of data is integer. Note how commands are tagged with control points (^{c}). Every function's body has also a final control point that signals the end of the function. These tags let us track the progress of program execution and are necessary to formally define the semantics of Newspeak.

Variable stack and store The memory is modeled with a stack and a store.

The **store** is composed of a collection of disjoint memory blocks. Each block is a sequence of consecutive bytes (in Byte) identified by a location ℓ (in Loc). The address (in Addr) of the o^{th} byte in block ℓ is thus denoted (ℓ, o). Within a block, address arithmetic is possible $(\ell, o+1)$ and $(\ell, o-1)$, being respectively the address following, and preceding (ℓ, o). However, two addresses from distinct

types
$\tau ::= \beta$ scalar type
$\beta ::= \mathbf{int}_n^\varepsilon$ integer

left values
$lv ::= x^-$ local variable

constants
$k ::= i$ integer
$fn ::= f_i$ function name

operators
$op ::= \mathtt{coerce}_I$ integer coercion
$\mid +, -, \times, /, \%$ integer arithmetic

expressions
$e ::= k$ constant
$\mid lv_\beta$ left value
$\mid op(e)$ unary operation
$\mid op(e_1, e_2)$ binary operation

commands
$cmd ::= {}^c lv =_\beta e$ scalar assignment
$\mid {}^c\{\tau;\ cmd^{c'}\}$ variable declaration
$\mid {}^c fn()$ function call
$\mid cmd_1 \dots cmd_n$ sequence

program
$prg ::= (f_1 = {cmd_1}^{c_1}, \dots, f_n = {cmd_n}^{c_n})$

Fig. 1. Syntax of a subset of Newspeak

$$\mathsf{Addr} \subseteq \mathsf{Loc} \times [0; N[\qquad \sigma \in \mathsf{Addr} \to \mathsf{Byte}$$
$$\sigma_n(\ell, o) = \sigma(\ell, o) \dots \sigma(\ell, o + n - 1)$$
$$\sigma[(\ell, o) \overset{n}{\mapsto} b_0 \dots b_{n-1}](a) = \begin{cases} b_i & \text{if } a = (\ell, o + i) \\ \sigma(a) & \text{otherwise} \end{cases}$$

Fig. 2. Store domain and accesses

blocks are incomparable. A sequence b of n bytes may be written to or read from the store at address a. To that end, we have the usual notations $\sigma[a \overset{n}{\mapsto} b]$ and $\sigma_n(a)$, formally defined in figure 2. When clear from context, the number of bytes n will be omitted.

In order to interpret a variable, it is necessary to know which memory block stores its value. The **variable stack** ρ provides this link. It maps an integer (the position of the variable in the stack) to a location in the store.

Data Programs manipulate scalar values, i.e. integers. Integer types have a sign parameter ε, to specify whether the type is signed (s) or unsigned (u), and a size n which represents the number of bytes needed for their binary encoding. All scalar types are assimilated to the set of values they represent, hence:

$$\mathbf{int}_n^s = [-2^{8n-1}; 2^{8n-1}[\qquad \mathbf{int}_n^u = [0; 2^{8n}[$$

A scalar value is stored in σ as a sequence of bytes. Hence, any scalar value can be encoded in bytes. Conversely, a sequence of bytes can (sometimes) be interpreted as a value. To that end, let us define a total function $\mathsf{tobytes}_\beta$ and a partial

function $\mathsf{ofbytes}_\beta$. It is unnecessary to precisely define these functions. Their behavior is, as a matter of fact, architecture-dependent, because, for example, of big or little endian byte ordering. These encoding and decoding functions should still obey a few equational rules, such as:

$$\mathsf{ofbytes}_\beta(\mathsf{tobytes}_\beta(v)) = v \tag{1}$$

The number of bytes used to encode a value of type τ is called the size of the type, and will be noted $|\tau|$. As for now, only $|\mathbf{int}_n^\varepsilon| = n$ is defined.

Our memory and data model is very similar in essence to the one found in [2] or [14]. However, in our case, the byte encoding of values, rather than the crude values themselves are stored in memory. This makes the model both lower level and simpler.

Left values and expressions In a stack ρ and a store σ, a left value lv and an expression e respectively evaluate to an address a and a value v. This is given by the two judgments $\rho, \sigma \vdash lv : a$ and $\rho, \sigma \vdash e : v$ inductively defined on the syntax.

The location of the memory block for a local variable x^-, is found at offset x from the top of the stack ρ. Assuming the height of the stack is n, it is thus equal to $\rho(n - x)$. Then the address of variable x is the address of the first byte in this block, hence:

$$\frac{n = |\rho| \qquad \ell = \rho(n - x)}{\rho, \sigma \vdash x^- : (\ell, 0)}$$

A constant value immediately evaluates to itself. To read the value of type β found in memory at a given address a, the sequence of $|\beta|$ bytes is first retrieved and then converted, if possible, with function $\mathsf{ofbytes}_\beta$:

$$\frac{}{\rho, \sigma \vdash k : k} \qquad \frac{\rho, \sigma \vdash lv : a \quad n = |\beta| \quad v = \sigma_n(a) \quad v \in \mathsf{dom}(\mathsf{ofbytes}_\beta)}{\rho, \sigma \vdash lv_\beta : \mathsf{ofbytes}_\beta(v)}$$

To evaluate an operator-based expression, the subexpression(s) is (are) first recursively evaluated then the operator applied (see appendix C for a formal definition). Since Newspeak expressions have no side-effects whatsoever, the order of evaluation does not matter. Since we did not introduce function pointers yet (see section 2.3), the semantics of a function name is straightforward: it evaluates to itself.

$$\frac{}{\rho, \sigma \vdash f_i : f_i}$$

Operators Arithmetic operators $+$, $-$, $\times$, $/$ and $\%$ are all standard natural number operations. For division, when the divisor is 0, it is an error that blocks the semantics. Similarly, for the congruence operator, $i\%0$ does not give any result. The formal definitions of these operators is systematic and can be found in appendix C.

Note that, unlike C arithmetic, these Newspeak operators are defined however large their results may be. In C, on most architectures, all integer arithmetic

is performed modulo. On some architectures, however, the behavior is less orthodox. In any case, Newspeak has an additional operator to mimic overflows: $\texttt{coerce}_I$. If its argument is not within the range I, then an arbitrary number in I is returned, otherwise this operation is silent. As a benefit, $\texttt{coerce}_I$ can also be used to model casts between integers of different size and/or sign. Section 3.3 explains the translation of C arithmetics and casts using coercion.

$$\frac{i \in I}{\texttt{coerce}_I(i) : i} \qquad \frac{i \notin I \qquad i' \in I}{\texttt{coerce}_I(i) : i'}$$

State An interpreter that runs Newspeak code is formally defined by a relation $\rightarrow$ in the style of operational semantics [16]. The notation $s_1 \rightarrow s_2$ describes the fact that the interpreter may perform an elementary execution step from state s_1 to state s_2. A state s of the interpreter is a tuple (c, ρ, σ, D, S). In addition to the stack ρ and store σ, the state has three other components:

- A **control point** c tracks the current position of the interpreter relative to the program code,
- A **location dump** D collects all the locations ever created since the beginning of execution. It allows to generate a fresh location, distinct from any other whenever a new memory block is created,
- At last, a **return stack** S piles the control points of the instruction to return to once a function call ends.

This program state is very close to the one described in [17], except that the location dump is encoded in the state (thanks to counters). It would thus be easy to add dynamic allocation as a core primitive to Newspeak while keeping the same semantic state.

Commands We note ${}^{c}cmd^{c'}$ the fact that control point c labels the first instruction of the command cmd and that c' is the next control point according to the control flow graph of the program. The formal definition of this notation is straightforward, yet painstaking. For sake of completeness, it is recalled in appendix B.

The operational semantics is then given by a few inference rules. When the interpreter reaches an assignment, it first evaluates the left value as an address a. It then proceeds to evaluate the right value. The bytes that represent this value, seen as having type β, are then written in the store starting at address a:

$$\frac{{}^{c}lv =_\beta e^{c'} \qquad \rho, \sigma \vdash lv : a \qquad \rho, \sigma \vdash e : v}{(c, \rho, \sigma, D, S) \rightarrow (c', \rho, \sigma[a \mapsto \mathsf{tobytes}_\beta(v)], D, S)}$$

At local variable declaration, a fresh store location ℓ is created and pushed on top of the stack ρ. Simultaneously, a new block of memory is allocated in the store at ℓ and initialized with random bytes. When execution reaches the end of the valid scope of the variable, the top of the stack is popped and the corresponding

memory block freed.

$$\frac{{}^{c}\{\tau;\ {}^{c'}cmd\} \qquad n = |\rho| + 1 \qquad \ell \notin D \qquad v \in \mathsf{Byte}^{|\tau|}}{(c, \rho, \sigma, D, S) \to (c', \rho[n \mapsto \ell], \sigma[(\ell, 0) \mapsto v], D \cup \{l\}, S)}$$

$$\frac{\{\tau;\ cmd^{c}\}^{c'} \qquad n = |\rho| - 1 \qquad \rho' = \rho_{|[1;n]} \qquad \sigma' = \sigma_{|\{(\ell,o)|\ell \neq \rho(n+1)\}}}{(c, \rho, \sigma, D, S) \to (c', \rho', \sigma', D, S)}$$

At function call, the interpreter evaluates the function name and redirects execution to the first control point of the function. It also pushes the control point of the command that follows the call on top of the return stack. So that, when execution of the function ends, execution may continue:

$$\frac{{}^{c}fn()^{c'} \quad \rho, \sigma \vdash fn : f_i \quad f_i = {}^{c_i}cmd_i}{(c, \rho, \sigma, D, S) \to (c_i, \rho, \sigma, D, S.c')} \qquad \frac{f_i = cmd_i{}^{c}}{(c, \rho, \sigma, D, S.c') \to (c', \rho, \sigma, D, S)}$$

Similarly to an assembly language, Newspeak function call has no arguments. As shown in 3.4 in the translation of C function calls, it is up to the caller to push them on the stack and up to the callee to read them at the right position in the stack according to calling conventions.

Note that there are no rules for the sequence of commands, as they are implicitly given by the control flow graph.

2.2 Composite data structures

There are two composite data structures: arrays and regions.

$$\tau \quad ::= \quad \ldots \ \mid \ \tau[n] \ \mid \ \{\tau_1\ o_1; \ldots;\ \tau_k\ o_k\}_n$$

An array $\tau[n]$ is a sequence of n consecutive elements of type τ. A region $\{\tau_1\ o_1; \ldots;\ \tau_k\ o_k\}_n$ is a block of n consecutive bytes. A region indicates which kind of data may be expected at offsets o_1 to o_k from the start of the block. Naturally, all values must be contained within the region, so we always have:

$$\forall i : 0 \leq o_i \leq n - |\tau_i|$$

In order to access a specific element of an array, or field of a region, the address of the block is first computed, and then incremented by some offset. Newspeak's *shift* construction + serves this exact purpose:

$$lv \quad ::= \quad \ldots \ \mid \ lv + e \qquad\qquad \frac{\rho, \sigma \vdash lv : (\ell, o) \qquad \rho, \sigma \vdash e : i}{\rho, \sigma \vdash lv + e : (\ell, o + i)}$$

Newspeak provides the operator $\texttt{belongs}_I$ that blocks the execution when its argument does not belong to the range I:

$$op \quad ::= \quad \ldots \ \mid \ \texttt{belongs}_I \qquad\qquad \frac{i \in I}{\texttt{belongs}_I(i) : i}$$

This operator is especially useful to explicit array bounds checks, and will be illustrated in the first example of section 3.5.

The region copy allows to copy a whole block of memory from some address to another address:

$$cmd \quad ::= \quad \ldots \mid {}^{c}lv_1 =_n lv_2$$

$$\frac{{}^{c}lv_1 =_n {lv_2}^{c'} \qquad \rho, \sigma \vdash lv_1 : a_1 \qquad \rho, \sigma \vdash lv_2 : a_2}{(c, \rho, \sigma, D, S) \rightarrow (c', \rho, \sigma[a_1 \mapsto \sigma_n(a_2)], D, S)}$$

Region copy is less powerful than the C standard library function *memcpy()*, since the number n of bytes to copy is statically determined. In practice, and as seen in section 3.4, this command is used to encode C structure or union assignment.

2.3 Pointers

Newspeak has only two kinds of pointers: pointers on data and pointers on function. The size of pointers (|**ptr**| and |**fptr**|) is left as a parameter, since it may vary according to the architecture.

$$\beta \quad ::= \quad \ldots \mid \textbf{ptr} \mid \textbf{fptr}$$

Newspeak pointer types do not carry any information about the kind of value they reference. Indeed, in C, it is valid to cast any kind of data pointer to another data pointer. This is particularly true of casts from and to the `char*` or `void*` types. Casts between pointers are useful to implement a weak form of polymorphism (such as in the *memcpy()* function). However, this permissivity makes the indication on pointer types unreliable: a static analysis cannot rely on the C type information to infer the kind of data that may be accessed from a given pointer. As a matter of fact, in C, *any sequence of bytes may be interpreted as any kind of data that fits.* As a side-effect, casts between pointers are absent from Newspeak.

Newspeak has syntax to perform pointer creation, dereference and arithmetic:

$$\begin{array}{rclcrcl} lv & = & \ldots \mid [e]_k & & fn & = & \ldots \mid [e]_{\tau_1 \ldots \tau_k \rightarrow \tau} \\ e & = & \ldots \mid \texttt{nil} \mid \&_n lv \mid \&_{\tau_1 \ldots \tau_k \rightarrow \tau} f & & op & = & \ldots \mid +_p \mid -_p \end{array}$$

A data pointer is either null (`nil`) or a tuple $\langle (\ell, o), [o_1; o_2] \rangle$, in which case it refers to the address (ℓ, o). This pointer is valid as long as it stays, or accesses, data within the memory zone that stretches from (ℓ, o_1) to (ℓ, o_2).

$$\textbf{ptr} = \{\texttt{nil}\} \cup \{\langle (\ell, o), [o_1, o_2] \rangle \mid (\ell, o) \in \mathsf{Addr} \wedge 0 \leq o_1 \leq o \leq o_2 \leq N\}$$

Hence, such a pointer may be created from a left value lv and a size n. Then, pointer arithmetic may shift the address (l, o). Yet, program execution stops with a pointer out of bounds whenever o leaves the memory zone delimited by o_1 and o_2. At last, a pointer may be dereferenced to access k bytes of data. However, it is an error if either the block referred to by the pointer has been deallocated (ℓ is

not in the domain of the store anymore), or the valid zone of the pointer can not hold all k bytes ($o + k > o_2$). Whenever two pointers refer to different addresses within the same memory block, the distance between them can be computed. Pointer arithmetic and dereference is invalid on the null pointer. Formally:

$$\frac{\rho, \sigma \vdash lv : (l, o)}{\rho, \sigma \vdash \&_n lv : \langle (l, o), [o; o + n] \rangle} \qquad \frac{o' = o + i \qquad o_1 \leq o' \leq o_2}{\langle (l, o), [o_1; o_2] \rangle +_p i : \langle (l, o'), [o_1; o_2] \rangle}$$

$$\frac{\rho, \sigma \vdash e : \langle (l, o), [o_1; o_2] \rangle \qquad \ell \in \mathsf{dom}(\sigma) \qquad o + k \leq o_2}{\rho, \sigma \vdash [e]_k : (l, o)}$$

$$\frac{}{\langle (l, o_1), r_1 \rangle -_p \langle (l, o_2), r_2 \rangle : o_1 - o_2}$$

A function pointer is either null or created from a function name. Formally, we have $\mathbf{fptr} = \{\mathtt{nil}\} \cup \{f_1, \ldots, f_n\}$. When dereferenced, the function's type and expected type at call site are checked for equality:

$$\frac{}{\rho, \sigma \vdash \&f : f} \qquad \frac{\rho, \sigma \vdash e : f \qquad f : \tau_1 \ldots \tau_k \to \tau}{\rho, \sigma \vdash [e]_{\tau_1 \ldots \tau_k \to \tau} : f}$$

2.4 Type conversions

We have seen in 2.1 how coercion stands for the cast between various kinds of integers, and mentioned in 2.3 that casts between pointer types are transparent. Any other cast amounts to the conversion of the value v into a sequence of bytes and then back to a value of the target type β_2. Note that this behavior is the same as if the value were first written to the store to be later read as a value of type β_2.

$$op \quad = \quad \ldots \mid (\beta_1 \rhd \beta_2) \qquad \frac{v' = \mathsf{tobytes}_{\beta_1}(v) \qquad v' \in \mathsf{dom}(\mathsf{ofbytes}_{\beta_2})}{(\beta_1 \rhd \beta_2)(v) : \mathsf{ofbytes}_{\beta_2}(v')}$$

In C, the integer 0 is always interpreted as the null pointer. This puts an additional constraint on the encoding/decoding functions:

$$\mathsf{tobytes}_{\mathbf{int}_n^\varepsilon}(0) = \mathsf{tobytes}_{\mathbf{ptr}}(\mathtt{nil}) \qquad \text{when } n = |\mathbf{ptr}|$$
$$\mathsf{tobytes}_{\mathbf{int}_n^\varepsilon}(0) = \mathsf{tobytes}_{\mathbf{fptr}}(\mathtt{nil}) \qquad \text{when } n = |\mathbf{fptr}|$$

Note, that together with (1), and the semantics of casts, this rule immediately implies that $(\mathbf{ptr} \rhd \mathbf{int}_n^\varepsilon)(\mathtt{nil})$ yields 0. As a sidenote, let us mention that not every sequence of bytes encodes a valid pointer. Indeed, if a program has no pointer declaration, there isn't any sequence of bytes that represents a pointer! That is why, in general, function $\mathsf{ofbytes}_{\mathbf{ptr}}$ is partially defined.

2.5 Control commands

Selection So as to offer choices between different branches of executions, Newspeak includes à la Dijsktra guarded alternative [5]:

$$cmd \quad ::= \quad \ldots \mid {}^c\Sigma_{0<i\leq n}\ (b_i) \to cmd_i \qquad b \quad ::= \quad e_1 \ldots e_k$$

The interpreter executes one of the commands associated to one of the guards that holds in the current state of memory. A guard $b = e_1 \dots e_k$ holds if none of its expressions evaluates to 0:

$$\frac{{}^c\Sigma_{0<i\leq n}\ (b_i) \rightarrow {}^{c_i}\, cmd_i \quad \rho, \sigma \vdash b_j}{(c, \rho, \sigma, D, S) \rightarrow (c_j, \rho, \sigma, D, S)}$$

$$\frac{\rho, \sigma \vdash e_1 : v_1 \quad \dots \quad \rho, \sigma \vdash e_k : v_k \quad \forall i : v_i \neq 0}{\rho, \sigma \vdash e_1 \dots e_k}$$

Newspeak guards explicitly provides conjunction; disjunction may be encoded with various alternatives. With the negation, equality and inequality operators, all kinds of boolean formulas may be expressed in Newspeak. Both equality and inequality carry the kind of scalar value they compare. Data pointers are compared by their offsets, only when they refer to the same memory block. Inequality is not defined for function pointers. Except for pointer inequality, the other boolean operators have standard semantics that need no further explanation:

$$op \quad ::= \quad \neg \ \mid \ \geq_\beta \ \mid \ ==_\beta$$

$$\frac{o_1 \geq o_2}{(\ell, o_1) \geq_{\mathbf{ptr}} (\ell, o_2) : 1} \qquad \frac{o_1 < o_2}{(\ell, o_1) \geq_{\mathbf{ptr}} (\ell, o_2) : 0}$$

Loop and structured forward gotos Newspeak has only one kind of loop, the infinite loop, that runs forever (see appendix C for its formal semantics). There needs to exist some way to break out of an infinite loop. This is the role of structured forward gotos. Indeed, `do` *cmd* `with` *lbl*: construct let us name a block *cmd* with a label *lbl*. It then becomes possible to jump from any place inside this block directly to the end label *lbl*:

$$cmd \quad ::= \quad \dots \quad \dots \ \mid \ {}^c\texttt{forever}\ cmd \ \mid \ \texttt{do}\ cmd\ \texttt{with}\ lbl\texttt{:} \ \mid \ {}^c\texttt{goto}\ lbl$$

$$\frac{{}^c\texttt{goto}\ lbl \qquad \texttt{do}\ cmd\ \texttt{with}\ lbl\texttt{:}^{c'}}{(c, \rho, \sigma, D, S) \rightarrow (c', \rho, \sigma, D, S)}$$

It is assumed that a goto and its target label are within the same variable scope. Otherwise some variables may never be popped from the local stack and subsequent variables accesses would be erroneous. Moreover, all labels defined in a program are distinct. Note also that `do` *cmd* `with` *lbl*: has no control point of its own: this instruction is really nothing more than an annotation.

These constructions can encode C loop exit, break, continue and even multiple returns in a function, see section 3.4 for an example of translation. However backward or intertwined gotos (`goto l1; goto l2; l1: ... l2:`) are not trivially representable in Newspeak. The choice of control structures was motivated by the necessity to keep a static analysis algorithm both simple and efficient. Forward gotos may be handled with a forward pass using a table to join all information related to a given label. And infinite loops clearly isolate fixpoint computations.

As a last remark, forward gotos and do with commands are syntactically very similar to exception raise and catch. They may easily be extended in the

future, if other languages than C (for instance Ada [8]) need to be compiled into Newspeak.

2.6 Globals, initial state

A Newspeak program consists in a list of globals with their initializations, and a list of functions with their bodies, one of them being the main function.

$$prg ::= (\tau_1\ g_1 = init_1, \ldots, \tau_k\ g_k = init_k; f_1 = {cmd_1}^{c_1}, \ldots, f_n = {cmd_n}^{c_n})$$
$$init ::= zero \mid \{o_1 =_{\beta_1} e_1; \ldots o_k =_{\beta_k} e_k\}$$

At startup, the initial store σ_0 contains a memory block for each global variable. As defined in appendix C, these blocks are, by default, filled with 0. However, they can also be explicitly initialized with static expressions. A global will be directly accessed by its name:

$$lv ::= \ldots \mid g \qquad \frac{}{(\rho, \sigma) \vdash g : (g, 0)}$$

Execution starts at the first control point c_0 of the main function in a state $(c_0, \epsilon, \sigma_0, \{g_1 \ldots g_k\}, \epsilon)$, where both local variable and return stack are empty.

This ends our exposition of the formal semantics of Newspeak. The only aspects of the language we didn't address are bitwise operators (boolean and shift), and floating point operations. Bitwise operators are fairly straightforward. On the contrary, floating points operations are complex and necessitate to study other standards such as [10]. Newspeak may still evolve in this regard.

3 Translation from C to Newspeak

The compilation from C to Newspeak preserves the semantics of the source programs, as we understood it from the informal definition made by the ANSI C standard [7]. In particular, we often compared C2newspeak output to the behavior of the corresponding C file compiled.

3.1 Type translation

The main problem concerning types is that they depend on the architecture. The user can describe its architecture by way of some parameters. In all our example, we will assume the architecture to be i386.

Sizes of integer types are explicited by the translation. Declarations such as **int** and **unsigned char** would be translated into $\mathbf{int}_4^s$ and $\mathbf{int}_1^u$.

Pointer types are even simpler, because we immediately forget the type of data they are supposed to point to. The only distinction made is between data and function pointers.

`int*`	$\rightsquigarrow$	**ptr**
`char**`	$\rightsquigarrow$	**ptr**
`int (*)(int)`	$\rightsquigarrow$	**fptr**

The offset (in number of bytes) of field f in **struct** T is computed by an architecture-dependent function, $\mathsf{offsetof}_{\mathsf{T}}(\mathsf{f})$ during translation. The computation of this offset may depend on the underlying architecture and on the alignment/packing policy employed. We see that C structures and unions are both translated into Newspeak regions, the difference being that union fields are all at offset 0.

```
struct {                 {
  char* x;       ⇝         ptr 0
  char y;                  int_1^s 4
  int z;   } s;            int_4^s 8 }_12
union {                  {
  int i;         ⇝         int_4^s 0
  char c;  } u;            int_1^s 0 }_4
```

Finally, an array **short**[20] will be translated into the Newspeak type $\mathbf{int}_2^s[20]$.

3.2 Variables

Here is a simple program that declares a global variable x. It then adds two local variables and performs some assignments which shows how local variables are pushed on a stack. To lighten the representation of the scopes, we will assume all the variables used from now on are global ones.

```
int x;
void main() {      main() {
  int y;             {int_4^s
  int z;       ⇝     {int_4^s
  x = y;             x =_int_4^s 1^-_int_4^s
  x = z; }           x =_int_4^s 0^-_int_4^s }}}
```

3.3 Left values and expressions

Additions, as well as other arithmetic operations, have to be translated as the composition of two Newspeak operations: the addition of natural integers, followed by the explicit integer coercion into the appropriate type. *($[m, M[$ stands for the domain of $\mathbf{int}_4^s$: $[-2^{31}, 2^{31}[$).*

```
int x,y,z;
x = y + z;   ⇝  x =_int_4^s coerce_[m,M[(y_int_4^s + z_int_4^s)
```

Here is a cast from a 4-byte integer into a 1-byte integer. The translation provided by C2newspeak and the behavior of the coerce operator (seen in section 2.1) ensures the soundness with respect to the C standard: if x is out of the 1-byte integer's scope, y will take a random value in its domain.

```
int x; unsigned char y;
y = x;  ⇝  y =_int_1^u coerce_[0,2^8[(x_int_4^s)
```

3.4 Statements

C assignment can be translated by two different Newspeak primitives: the scalar assignment and the region copy. The example with a and b, two variables of type **struct** {**int** x; **int** y} shows this distinction. We may notice that the shift (+) construction used here would also translate array elements access, as shown in the first example of section 3.5.

```
struct {int x; int y} a,b;
a.x = b.y;  ⇝  a + 0 =_int_4^s (b + 4)_int_4^s
a = b;      ⇝  a =_8 b
```

A **while** loop becomes an infinite loop wrapped into a `do ... with` *lbl*: block. In the loop, the condition is then checked with a guarded alternative at the beginning of each iteration, using a forward goto to *lbl* to explicitly break the control flow. We have here an application of the design rule of minimality: a C instruction is split into many Newspeak primitives, which can be reused to translate other statements (**for**

```
x = 0;                    x =_β 0;
while (x < 10)  ⇝         do { forever {
  x++;                      | (x ≤_β 10) → goto lbl
                            | (¬x ≤_β 10) → ∅
                            x =_β coerce_[m,M[(x_β + 1)
                          }} with lbl:
```

loops, **if** and **switch** choices). *(β stands for $\boldsymbol{int}_4^s$, and $[m, M[$ for $[-2^{31}, 2^{31}[$).*

Newspeak functions use the same conventions as in assembler: it is the duty of the caller to prepare the variable stack. In our example, f expects 0^- to be the argument x and 1^- to be a variable ready to receive its result.

```
char f (char x)
 {return (x+1);}
```
$\rightsquigarrow$
$f = \{1^- =_{\mathbf{int}_1^s}$
$\quad \mathtt{coerce}_{[-2^7,2^7[}(0_\beta^- + 1); \}$

When the function is called, the return value is first declared, then the argument is declared and initialized; thus, the top of the variable stack is ready for f. It is another example of the minimality rule, because we reuse the assignment and the declaration statements to translate arguments passing during function calls.

```
f (5);
```
$\rightsquigarrow$
$\mathbf{int}_1^s;\quad (value\_of\_f)$
$\mathbf{int}_1^s;\quad (x)$
$0^-_{\mathbf{int}_1^s} =_{\mathbf{int}_1^s} 5;$
$f();$

3.5 Generation of useful annotations for static analysis

When an array is accessed using a non-constant index, there might be a buffer overflow during the execution. C2newspeak uses the $\mathtt{belongs}_I$ operator to explicit the verification that has to be done by an analyzer. We will discuss in 4.1 a case where I can not be determined at compilation time.

```
int t[10];
int i,j;
j = t[i];
```
$\rightsquigarrow$ $j =_\beta \left(t + \mathtt{belongs}_{[0,9]}(i_\beta) \times 4\right)_\beta ;$

The manipulation of x and t use once again the shift operator $+$. We also discover here the "address of" operator & and the pointer dereference $[\cdot]$. Let's notice that both have annotations useful for implementing pointer checks during static-analysis: the first one to remember how many bytes can be addressed from the pointer created, and the second one to explicit the size of the area dereferenced.

```
int* x;
int t[100];
x = &t[3];
x = x + 5;
*x = 3;
```
$\rightsquigarrow$
$x =_{\mathbf{ptr}} \&_{400} t + (3 \times 4);$
$x =_{\mathbf{ptr}} x_{\mathbf{ptr}} + (5 \times 4);$
$[x_{\mathbf{ptr}}]_4 =_{\mathbf{int}_4^s} 3;$

4 Implementation

4.1 Organization of the compiler

Our compiler is implemented in OCaml [11]. It proceeds in several steps. We use CIL [15] to parse C programs and start with a CIL syntax tree. Several problems must then be overcome in order to get to a Newspeak file.

The first difficulty arises from the treatment of the CIL structure. Some artifacts introduced by CIL must be undone (for instance gotos and labels generated in **if** statements). Constant string variables must be considered as pointers on global initialized arrays. Static variables must be transformed in global variables and renamed to avoid name clashes. All these changes are local to each C file and are regrouped in the first pass.

For a project with several source files performance becomes an issue: compiling and merging thousands of files containing each tens of thousands of lines altogether is very memory-consuming. Furthermore, it can be necessary to analyze a group of files separately from the rest of the project, because the whole program source code is not available. That is why we have decided to use the same architecture as a classic C compiler, that is to say, a compiling pass, and a linking pass.

The compilation takes a preprocessed C file, performs the actual translation, possibly letting some blanks in variables or function definitions, that will have to be filled during the linking phase.

For instance, such a blank is introduced when an extern global array, say **extern int** myarray[] is declared in some file `a.c` and its size defined in another file `b.c`. If myarray is accessed in `a.c`, then the array bound check can not be generated before the linking phase.

Let us stress the fact that C2newspeak handles here a problem that `gcc` refuses. In fact, the program in the margin is rejected by `gcc` as it cannot compute the size of a.

```
extern int a[];
int main (void) {
  return (sizeof (a));
}
```

4.2 C2newspeak usage and performance obtained

By default, C2newspeak applies stringent restrictions on the C it inputs. For instance, casts between pointers and integers, or multiple definitions of the same variable are rejected. These protections are lifted if options `--castor` or `--accept-mult-def` are respectively set.

Some compilers also don't respect the standard fully, and won't initialize the global variables with zeros, so we added an option (`--no-init`) allowing to skip this initialization during the compilation.

Finally, C2newspeak can take C files to produce compiled object (with the `.no` extension), and then take either C and Newspeak object files to link a complete Newspeak program. It is possible to debug every phase of the translation. For example,

```
./c2newspeak --npko --cil --newspeak a.c b.c
```

will print CIL output and produce Newspeak object for both `a.c` and `b.c` files before printing the final result on standard output.

With the right options, we managed in the end to compile a 3.2-million-line program in one hour and a half, on a standard personal computer.

4.3 Other tools

Some additional utilities have been developed with C2Newspeak:

`npkstats` produces some statistics about a Newspeak program,

`npkstrip` takes a Newspeak program and extracts the global variables and functions necessary to the execution of *main*.

`npksimplify` performs some simplifications on Newspeak code.

All our programs are available at http://www.penjili.org/newspeak.html with the C2newspeak compiler and are distributed under the LGPL.

5 Related work

There exists other intermediate languages developed for the transformation and analysis of C software, such as CIL [15], Elsa [13] and C transformers [3]. Contrarily to Newspeak, these representations can be immediately output as a syntactically valid C program. For that, they all use syntax trees still very close to C, and so too complex and high level for our needs. However CIL is used in C2Newspeak as a first pass to perform the particularly ungrateful task of parsing and disambiguating C syntax.

Newspeak is closer to a high-level assembly language, and may be compared to languages such as the Java bytecode [12] or Microsoft's Common Intermediate Language (MS CIL)[9]. However these languages are far from concise (MS CIL has more than 100 different instructions). They have unstructured control flow (forward and backward jumps). In addition, both are three-address codes, meaning that complex expressions are broken down into a succession of individual operations. This may increase the cost of analysis, in particular those that are sensitive to the number of variables. Both bytecodes include constructions for object oriented features that are unnecessary for C.

The language closest to ours is probably Cminor [1] used in project Compcert. Compcert is a certified compiler developed in Coq [4]. Cminor's primitives capture the same language paradigm as Newspeak but slightly differs. For instance, Cminor's "`exit` n" instruction, indicates the number of enclosing blocks to skip, rather than give the name of a label to jump to. Cminor is not minimal. For instance, it has both local binding and local variables with whole function scope; both conditional expressions and conditional statements. Its functions have explicit arguments and return a value, which necessitates an additional return statement. Cminor memory model [2] is very close to ours, albeit slightly higher level: values rather than sequences of bytes are stored in the memory. Maybe, more importantly, Cminor was designed for compilation, and so lacks annotations needed by a static analyzer to perform checks.

Caduceus [6] is a verification tool for C that allows to use proof assistants to prove the correctness of C programs. It translates C into Why's formalism. The Why language is ML-like, and thus far from the essence of imperative languages. Indeed, Caduceus does not handle pointer casts, which is prohibitive when analyzing embedded software.

6 Conclusion, possible extensions, future work

The amount of work to accomplish in order to analyze real programming languages can be discouraging. We presented the syntax and semantics of a language designed for static analysis. As much as possible, we tried to make Newspeak concise and simple, yet expressive enough to capture C paradigms fully. A compiler from C to Newspeak was implemented and is now provided under the LGPL. We hope it allows others to quickly implement their static analysis algorithms and experiment with real C programs.

Newspeak is still an ongoing development. In particular, the exact syntax and delineation of floating point operations is not fixed yet. Furthermore, if further ways to simplify and structure the language are found, it will evolve.

In the future, several utilities could be implemented for Newspeak. For instance, rewriting of Newspeak programs, so as to normalize them, would limit the sensitivity of some static analysis algorithms on syntax. Some tools could insert hints based on heuristics. Program transformation could lower the number of variables to improve the performances of subsequent analyses.

Ultimately, as an intermediate language, Newspeak could contribute to decouple the analysis algorithms from the source programming language. To that end, translations from other languages may be implemented. The obvious first choice would be ADA, which is relatively close to C. Still, it would necessitate to enrich Newspeak slightly, in particular with primitives that encode exception handling.

References

1. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM'06*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006.
2. S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *ICFEM*, pages 280–299, 2005.
3. A. Borghi, V. David, and A. Demaille. C-Transformers — a framework to write C program transformations. *ACM Crossroads*, 12(3), 2006.
4. The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, 2002.
5. E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *ICFEM'04*, pages 15–29.
7. International Organization for Standardization. *Programming Languages, C, International standard, ISO/IEC 9899:1990*. 1990.
8. International Organization for Standardization. *Ada Reference Manual, ISO/IEC 8652:1995*. 1995.
9. International Organization for Standardization. *Information technology – Common Language Infrastructure (CLI) Partitions I to VI, ISO/IEC 23271:2006*. 2006.
10. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, 1985.
11. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). The objective caml system release 3.10. Technical report, INRIA, France, 2007.
12. T. Linholm and F. Yellin. *The Java Virtual Machine Specification*. 1999.
13. S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *CC'04*, 2004.
14. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES'06*. ACM Press, 2006.
15. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. 2002.
16. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
17. A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *SAS'02*, volume 2477 of *LNCS*. Springer, 2002.

A Newspeak syntax

Newspeak types

$\beta ::= \mathbf{int}_n^{\varepsilon}$		integer
	$\mid$ **ptr**	pointer
	$\mid$ **fptr**	function pointer
$\tau ::= \beta$		scalar type
	$\mid \tau[n]$	array
	$\mid \{\tau_1\ o_1; \ldots;\ \tau_k\ o_k\}_n$	region

Newspeak left value and expressions

constants		
$k ::= i$		integer
	$\mid$ `nil`	null pointer
operators		
$op ::= +, -, \times, /, \%$		integer arithmetic
	$\mid$ $\texttt{coerce}_I$	integer coercion
	$\mid$ $\texttt{belongs}_I$	bound check
	$\mid +_p, -_p$	pointer arithmetic
	$\mid \neg$	negation
	$\mid \geq_\beta, ==_\beta$	comparison
	$\mid (\beta_1 \rhd \beta_2)$	type conversion
left values		
$lv ::= x^-$		local variable
	$\mid g$	global variable
	$\mid lv + e$	address shift
	$\mid [e]_k$	pointer dereference
expressions		
$e ::= k$		constant
	$\mid lv_\beta$	left value
	$\mid \&_n lv$	pointer creation
	$\mid \&_{\tau_1 \ldots \tau_k \to \tau} f$	function pointer creation
	$\mid op(e)$	unary operation
	$\mid op(e_1, e_2)$	binary operation
$b ::= e_1 \ldots e_k$		conjunction
$fn ::= f_i$		function name
	$\mid [e]_{\tau_1 \ldots \tau_k \to \tau}$	function pointer dereference

Newspeak program and commands

commands

$cmd ::= {}^{c}lv =_{\beta} e$	scalar assignment
$\mid {}^{c}lv_1 =_n lv_2$	region copy
$\mid {}^{c}\{\tau;\ cmd^{c'}\}$	variable declaration
$\mid cmd_1 \ldots cmd_n$	sequence
$\mid {}^{c}\Sigma_{0<i\leq n}\ (b_i) \rightarrow cmd_i$	alternative
$\mid {}^{c}$`forever` cmd	infinite loop
$\mid$ `do` cmd `with` lbl:	forward label definition
$\mid {}^{c}$`goto` lbl	jump
$\mid {}^{c}fn()$	function call

initialization

$init ::= zero$	fill with zeros
$\mid \{o_1 =_{\beta_1} e_1; \ldots o_k =_{\beta_k} e_k\}$	fill with values

program

$prg ::= (\tau_1\ g_1 = init_1, \ldots, \tau_k\ g_k = init_k; f_1 = {cmd_1}^{c_1}, \ldots, f_n = {cmd_n}^{c_n})$

B Newspeak labelling

The following diagram describes the construction of the control flow graph by induction on syntax.

$$\frac{(\tau_1\ g_1 = init_1, \ldots, \tau_k\ g_k = init_k; f_1 = cmd_1{}^{c_1}, \ldots, f_n = cmd_n{}^{c_n})}{cmd_i{}^{c_i}}$$

$$\frac{{}^{c_0}\{\tau;\ cmd^{c_2}\}}{cmd^{c_2}} \qquad \frac{{}^{c_0}\{\tau;\ cmd^{c_2}\} \qquad {}^{c_1}cmd}{{}^{c_0}\{\tau;\ {}^{c_1}cmd^{c_2}\}}$$

$$\frac{cmd_1 \ldots cmd_n{}^{c_n}}{cmd_n{}^{c_n}} \qquad \frac{cmd_1 \ldots cmd_n \qquad {}^{c_i}cmd_{i+1}}{cmd_i{}^{c_i}}$$

$$\frac{cmd_1 \ldots cmd_n \qquad {}^{c_0}cmd_1}{{}^{c_0}cmd_1 \ldots cmd_n}$$

$$\frac{({}^{c}\Sigma_{0<i\leq n}\ (b_i) \rightarrow cmd_i)\,{}^{c'}}{cmd_i{}^{c'}} \qquad \frac{{}^{c}\Sigma_{0<i\leq n}\ (b_i) \rightarrow cmd_i \qquad {}^{c_i}cmd_i}{{}^{c}\Sigma_{0<i\leq n}\ (b_i) \rightarrow {}^{c_i}cmd_i}$$

$$\frac{{}^{c}\texttt{forever}\ cmd}{cmd^{c}} \qquad \frac{{}^{c}\texttt{forever}\ cmd \qquad {}^{c'}cmd}{{}^{c}\texttt{forever}\ {}^{c'}cmd}$$

$$\frac{\texttt{do}\ cmd\ \texttt{with}\ lbl{:}^{c}}{cmd^{c}} \qquad \frac{\texttt{do}\ cmd\ \texttt{with}\ lbl{:} \qquad {}^{c}cmd}{{}^{c}\texttt{do}\ cmd\ \texttt{with}\ lbl{:}}$$

$$\frac{{}^{c}cmd \qquad cmd^{c'}}{{}^{c}cmd^{c'}}$$

C Newspeak missing semantic element

Size of types

$$
\begin{aligned}
|\mathbf{int}_n^\varepsilon| &= n \\
|\mathbf{ptr}| &= p_1 \\
|\mathbf{fptr}| &= p_2 \\
|\tau[n]| &= n \times |\tau| \\
|\{\tau_1\ o_1; \ldots;\ \tau_k\ o_k\}_n| &= n
\end{aligned}
$$

Semantic of unary and binary operations

$$\frac{\rho, \sigma \vdash e : v \qquad op(v) : v'}{\rho, \sigma \vdash op(e) : v'}$$

$$\frac{\rho, \sigma \vdash e_1 : v_1 \qquad \rho, \sigma \vdash e_2 : v_2 \qquad v_1\ op\ v_2 : v}{\rho, \sigma \vdash op(e_1, e_2) : v}$$

Semantic of operators

$$\frac{}{i_1 + i_2 : i_1 + i_2} \qquad \frac{}{i_1 - i_2 : i_1 - i_2} \qquad \frac{}{i_1 \times i_2 : i_1 \times i_2}$$

$$\frac{i_2 \neq 0}{i_1/i_2 : i_1/i_2} \qquad \frac{i_1 \geq 0 \quad i_2 > 0}{i_1 \% i_2 : i_1 \% i_2} \qquad \frac{i_2 \neq 0 \quad i_1 < 0 \vee i_2 < 0 \quad |i'| < |i_2|}{i_1 \% i_2 : i'}$$

When either of the congruence operands is strictly negative, the result is a number whose absolute value is less than the absolute value of the divisor.

Semantic of infinite loop

$$\frac{{}^c\texttt{forever}\,{}^{c'}\,cmd}{(c, \rho, \sigma, D, S) \rightarrow (c', \rho, \sigma, D, S)}$$

Note that there is no rule for the loop's back edge, since this is implicitly given by the control flow graph.

Semantic of initial state

$$\frac{n = |\tau|}{\sigma \vdash \tau\ g = zero : \sigma[(g, 0) \mapsto 0^n]}$$

$$\frac{n = |\tau| \quad b \in \mathsf{Byte}^n \quad \sigma' = \sigma[(g, 0) \mapsto b] \quad (\epsilon, \epsilon) \vdash e_i : v_i}{\sigma \vdash \tau\ g = \{o_1 =_{\beta_1} e_1; \ldots o_k =_{\beta_k} e_k\} : \sigma'[(g, o_i) \mapsto \mathsf{tobytes}_{\beta_i}(v_i)]_{0<i\leq k}}$$

$$\frac{\epsilon \vdash \tau_1\ g_1 = init_1 : \sigma_1 \ldots \sigma_{k-1} \vdash \tau_k\ g_k = init_k : \sigma_k}{\tau_1\ g_1 = init_1, \ldots, \tau_k\ g_k = init_k : (c_0, \epsilon, \sigma_k, \{g_1 \ldots g_k\}, \epsilon)}$$