Published in "Static Analysis Symposium" (SAS) 2005, pages 155-170.

Using Dependent Types to Certify the Safety of Assembly Code *

Matthew Harren and George C. Necula

Computer Science Division, University of California
Berkeley, CA, USA 94720-1776
{matth, necula}@cs.berkeley.edu

Abstract. There are many source-level analyses or instrumentation tools that enforce various safety properties. In this paper we present an infrastructure that can be used to check independently that the assembly output of such tools has the desired safety properties. By working at assembly level we avoid the complications with unavailability of source code, with source-level parsing, and we certify the code that is actually deployed.
The novel feature of the framework is an extensible dependently-typed framework that supports type inference and mutation of dependent values in memory. The type system can be extended with new types as needed for the source-level tool that is certified. Using these dependent types, we are able to express the invariants enforced by CCured, a source-level instrumentation tool that guarantees type safety in legacy C programs. We can therefore check that the x86 assembly code resulting from compilation with CCured is in fact type-safe.

1 Introduction

There are numerous ongoing efforts to design static analyses or instrumentation tools to ensure various safety and security properties of programs. In most cases, there is no independent way to ensure that the analysis or instrumentation tool was actually run on a given program. Since most of today's software security tools operate only on source code, a concerned user must obtain the source for the program in question, must run the tool himself, and is forced to trust that the tool and the compiler are working as advertised. In this paper, we describe our efforts to develop an independent verification strategy for static analyses and instrumentation tools.

A well-known example of the strategy that we advocate is the verification of type safety in Java and .NET bytecode. A compiler verifies that the original source code is type-safe, and uses this typing information to generate typed

* This research was supported in part by the National Science Foundation under grant number CCR-00225610, CCF-0524784, and CCR-0326577. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

bytecode. The bytecode can then be checked for safety independently from the source code. We want to push this strategy to lower-level languages, such as assembly, and to allow more language-based enforcement tools to make use of it. Working at the assembly-language level makes our technique fit well in the current standard object-code distribution process. Furthermore, it does not require the program source code, is applicable to more source languages, and eliminates the compiler from the trusted computing base.

An additional goal of our work is to make it relatively easy for tool writers to customize a generic certification infrastructure with the rules and invariants that should hold in the processed code. To this end, the certification infrastructure performs many operations that are likely to be needed across a variety of enforcement tools.

1.1 Motivation

This work was initially motivated by requests from CCured users to have independent verification that libraries or object files have been processed by CCured. CCured [1] is a source-to-source translator that guarantees type safety for legacy C code by inserting run-time checks before potentially unsafe operations. Where necessary, it modifies data structures to accommodate metadata such as array-bound information. CCured performs extensive static analysis to minimize the changes to data structures and the number of run-time checks necessary. CCured also has many different kinds of run-time checks, for arrays, pointers on the stack, or type hierarchies. A framework that can keep up with CCured's analysis and run-time checks would be suitable for certifying the result of simpler tools such as Cqual [2] and Stackguard [3]. We believe our framework is general enough to be used with languages other than C and safety policies other than type safety.

We cannot use standard Typed Assembly Languages [4] to encode the output of CCured for two main reasons. First, the instrumentation scheme used by CCured requires dependent types to encode, for example, that a field in a structure is a pointer to a memory area whose length is stored in another field. The DTAL [5] language is dependently-typed and is at the assembly-language level, but does not allow mutation of dependently-typed records. The ability to overwrite dependent memory locations is crucial for CCured, because most C programs store pointer values in memory. We propose in this paper a new dependently-typed language that allows mutable records, by allowing the dependent-type invariants to be temporarily broken inside a basic block.

The other major obstacle in using one of the existing typed assembly languages is that it would require a special compiler that produces the desired language. Instead, we want to apply this strategy even to source-to-source transformations, in which the output of the tool is compiled using an off-the-shelf compiler. The challenge posed by an external compiler is that register allocation and other optimizations will cause us to lose the correspondence between local variables in our source code and registers in the compiled code.

Our framework relies on (untrusted) annotations for function signatures and types of global variables. These annotations are generated by the source-level

tool whose policy we enforce. We decided against using such annotations for individual program points inside of a function's body, in order to reduce sensitivity to optimizations or compilation details. Instead, we use type inference to rediscover the types of the registers and stack slots in assembly code. Our use of abstract interpretation for type inference is similar to that used in bytecode verifiers, or to that described described by Chang et al. for compiler debugging [6]. For space reasons, we do not discuss type inference in this paper.

The contributions of this paper include:

- An expressive yet practical dependent type system for low-level code that supports mutable records. We describe in Section 2 the mechanism used for customizing the type system to new policies, and present the type system itself in Section 3.
- A description of the typechecking algorithm for this type system.
- An encoding of the safety constraints of CCured in this type system, with support for arrays, dynamic typing, and stack-allocated variables whose address is taken (Section 4). We describe in Section 5 our experience using a prototype verifier that can check the CCured output for type safety.

2 Type Policies

Our type system is parameterized by a *type policy* that describes the invariants enforced by the safety tool you wish to use (CCured, for example). Factoring our type system in this way provides modularity and allows us to support extension to different safety tools. Furthermore, it lets us focus this paper on the specific contributions of our framework, such as mutable dependent types and the infrastructure for type checking.

A type policy consists of the following:

- A finite set $\mathbb{T}$ of type constructors C. These constructors are used to build policy-specific types for word-sized values, as described below.
- A subtyping relation $\texttt{IsSubtype} : \tau \to \tau \to \mathit{Bool}$ for the types generated by these constructors, and the associated upper bound function $\texttt{TJoin} : \tau \to \tau \to \tau$ that returns a supertype of its arguments.
- An operation $\texttt{ArithType} : \tau \to \mathit{op} \to \tau \to \tau$ that assigns a type to the result of binary operators given the type of the operands, and an operation $\texttt{ConstType} : \mathit{const} \to \tau$ that gives a type to each constant.
- A `Constrain` operation that refines a typing context after a certain boolean expression has been tested to be true.

For example, a type policy could define a type constructor "Int" for integers that will fit in a machine word, and a constructor "MaybeNullPtr σ" for possibly-NULL pointers to records with type σ. We'll see below that the framework defines the "Ptr σ" type to describe pointers to σ. Then the policy will likely define both `IsSubtype`(Ptr σ, MaybeNullPtr σ) and `IsSubtype`(MaybeNullPtr σ, Int) to be `true`. Additionally, the policy might define `ArithType`(Ptr σ, "-", Ptr σ) to be

Int. Finally, the definition of `Constrain` for this policy may promote one or more values of type MaybeNullPtr σ to Ptr σ following an appropriate NULL-check.

We defer the more detailed discussion of the `IsSubtype`, `TJoin`, `ArithType`, and `Constrain` operators until the presentation of our typechecking algorithm in Section 3.1.

Although we currently trust the soundness of the type policy, our implementation is designed to facilitate formal proofs of the soundness of verification. Such a proof would rely on lemmas that the operators of the type policy are sound with respect to the definition of the type constructors.

3 Our Type System

We describe in this section our framework for dependent types, and show how a program can be typechecked with respect to a given type policy.

Figure 1 shows the language of memory types in our framework. Field types t describe the contents of a word in memory or in a register whereas σ types describe a mutable record consisting of a sequence of related fields.

field types	$t ::= C(d_1, \ldots d_n) \mid \text{Ptr } \sigma$
dependencies	$d ::= c \mid s.i \mid s$
record types	$\sigma ::= \texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$
constants	c
type constructors	$C \in \mathbb{T}$

Fig. 1. The types that are assigned to registers and memory locations.

The type of a word-sized location is either the instantiation of a type constructor C (given by the type policy) or a pointer to a mutable record. We saw above a few examples of nullary constructors for non-dependent types; constructors for dependent types are parameterized on one or more values. We distinguish the pointer type in our system so that we can give generic typing rules for memory reads and writes.

The notation $\texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$ denotes a very-dependent [7] record type with n mutable fields, each of whose types may depend on the runtime values of other fields. For simplicity, fields are labeled with their index in the record. The dependent type constructor "$\texttt{Rec}_s$" binds a variable s that can be thought of as the "self pointer" for the record. We use s to encode dependencies among the fields of the record: the special expression $s.i$ refers to the value stored in the i^{th} word of the current record, where i is a constant. We say that a field type $C(d_1, \ldots d_n)$ *refers to* field i iff at least one expression d_j is "$s.i$". A record type $\sigma = \texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$ is *well-formed* if for all terms $s.j$ referring to a field, we have $0 \leq j < n$. In other words, dependencies must refer to fields that actually exist. We require that all types used in this framework be well-formed.

For example, a type policy may define the singleton type constructor Single(e), and then can define a dependent record containing two identical integers as

$$\texttt{Rec}_s.\langle 0 : \text{Int};\ 1 : \text{Single}(s.0)\rangle$$

If we define the type constructor "Array(len)" to be the type of a pointer to an array of Ints with length len, then a record containing an array pointer and the length of that array has the type

$$\texttt{Rec}_s.\langle 0 : \text{Array}(s.1);\ 1 : \text{Int}\rangle$$

Field types can even refer directly to the self pointer s. $\texttt{Rec}_s.\langle 0 : \text{Single}(s)\rangle$ is a one-word object that contains a pointer to itself. Circular dependencies are also allowed, so

$$\texttt{Rec}_s.\langle 0 : \text{Single}(s.1);\ 1 : \text{Single}(s.0)\rangle$$

is another valid definition for our record containing two identical integers.

We therefore have two kinds of memory locations in the language. *Dependent* fields have types that refer to the self pointer or other fields, or are referred to by the types of sibling fields. *Non-dependent* fields have types of the form C (or $C(c_1, \ldots c_n)$, where each c_i is a constant) that do not refer to, and are not referred to by, any other field. We must be careful when a dependent field is updated, to ensure that the dependencies are respected. However, we can modify non-dependent fields in place without additional checking.

We also support dependent function types, including function pointers. Checking dependent functions is very similar to checking that dependent records are used correctly, and we do not discuss them further here.

3.1 Type checking

We describe here the process of typechecking assembly code when the start of each basic block has been annotated with an invariant, as is done in TAL [8]. For space reasons, we do not discuss in this paper our inference system for generating such invariants.

Figure 2 shows the simple MIPS-like assembly language that we will be typechecking. A basic block is a sequence of instructions whose entry is denoted by some label, and whose exit is a branch or a jump. Note that in this paper, we omit details relating to stack handling or the calling convention [9]. Our implementation uses the stack analysis engine written for the Open Verifier project [10].

We must track the memory state explicitly in order to reason about writes to dependent fields. "$\texttt{upd}(m, e_1, e_2)$" denotes the memory state that results from modifying memory state m by writing value e_2 at location e_1, while "$\texttt{sel}(m,\ e)$" is the result of reading address e in memory state m. We define "`ValidMem`" to be the type of a memory heap that is in a *consistent* state: one where all allocated locations contain a value that adheres to the type that the location was assigned when it was allocated. Consistency may be temporarily broken when we write a dependent field, since in general we will have to write to all of the fields in a dependent group before we can conclude that the group is consistent. But we will check that consistency holds at basic block boundaries.

instructions	$I ::= \texttt{mov}\ \mathrm{r}_{dest}, c \mid \texttt{mov}\ \mathrm{r}_{dest}, L \mid alu\ \ \mathrm{r}_{dest}, \mathrm{r}_{s1}, \mathrm{r}_{s2}$
	$\mid \texttt{load}\ \mathrm{r}_{dest},\ \mathrm{r}_a \mid \texttt{store}\ \mathrm{r}_{src},\ \mathrm{r}_a$
arithmetic	$alu ::= \texttt{add} \mid \texttt{mult} \mid \texttt{xor} \mid \texttt{slt} \mid \ldots$
labels	L
jumps	$J ::= \texttt{beq}\ \mathrm{r}_c, L \mid \texttt{jump}\ L \mid \texttt{jr}\ \mathrm{r}$
basic blocks	$B ::= I, B \mid J$
functions	$F ::= \langle L_1 : B_1,\ \ldots,\ L_m : B_m \rangle$

Fig. 2. The target assembly language.

states	$S ::= \langle \Delta, \Gamma, m \rangle$
register states	$\Delta ::= \mathrm{r}_1 = e_1,\ \ldots,\ \mathrm{r}_k = e_k$
type states	$\Gamma ::= v_1 \mapsto \tau_1,\ v_2 \mapsto \tau_2, \ldots$
memory states	$m ::= \texttt{upd}(m, e_1, e_2) \mid v$
abstract values	v
register types	$\tau ::= C(e_1, \ldots e_n) \mid \mathrm{Ptr}\ \sigma$
symbolic expressions	$e ::= c \mid v \mid L \mid \texttt{sel}(m,\ e) \mid e_1\ op\ e_2$
binary operations	$op ::= + \mid \times \mid \texttt{xor} \mid < \mid \ldots$

Fig. 3. The states of our symbolic execution algorithm for typechecking.

Our typechecker performs symbolic evaluation on one basic block at a time, using abstract values v for any unknown values. As seen in Figure 3, a state in our checker is $\langle \Delta, \Gamma, m \rangle$, where Δ is a mapping from registers to symbolic expressions, Γ is a mapping from abstract values to types, and m is the current memory state. We could represent a checker state in which r_2 was known to equal $\mathrm{r}_1 + 1$ as $\langle \Delta_0, \Gamma_0, v_{mem0} \rangle$, where:

$$\Delta_0 = \{\mathrm{r}_1 = v;\ \mathrm{r}_2 = v + 1\}$$
$$\Gamma_0 = \{v \mapsto \mathrm{Int}; v_{mem0} \mapsto \texttt{ValidMem}\}$$

This state can be considered syntactic sugar for the following logical formula:

$$\exists v \in \mathrm{Int}\,.\,\exists v_{mem0} \in \texttt{ValidMem}\,.\,(\mathrm{r}_1 = v) \wedge (\mathrm{r}_2 = v + 1)$$

Typing expressions We give here the rules for assigning types to symbolic expressions. The judgment $\Gamma \vdash e : \tau$ means that expression e has type τ in context Γ. Most of the this work is done by the type policies through the functions `ConstType`, `IsSubtype`, and `ArithType`, while the framework maintains the types of abstract variables and handles the typing of memory operations.

$$\frac{}{\Gamma \vdash v : \Gamma(v)}\ [\text{Abstract}] \qquad \frac{\tau = \texttt{ConstType}(c)}{\Gamma \vdash c : \tau}\ [\text{Const}]$$

$$\frac{\Gamma \vdash e : \tau' \qquad \texttt{IsSubtype}(\tau',\ \tau)}{\Gamma \vdash e : \tau}\ [\text{Subsumption}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau = \texttt{ArithType}(\tau_1, op, \tau_2)}{\Gamma \vdash e_1\ op\ e_2 : \tau}\ [\text{Arith}]$$

Type policies that do not care about arithmetic can say that $\texttt{ArithType}(\tau_1, op, \tau_2)$ is Int for all inputs, but policies such as CCured will derive a more precise type for some inputs to `ArithType`.

The final form of expression is a read from memory. When reading a dependent field with type $C(s.j)$, we must replace dependency $s.j$ with a symbolic expression that explicitly encodes the current value of the j^{th} field. Consider a record that contains an array pointer and its length, and suppose we read the array field into r_1 and the length field into r_2: [1]

$$\Delta = \{\mathrm{r}_1 = \texttt{sel}(m_0,\ v);\ \mathrm{r}_2 = \texttt{sel}(m_0,\ v+1)\}$$
$$\Gamma = \{v \mapsto \text{Ptr } \texttt{Rec}_s.\langle 0 : \text{Array}(s.1);\ 1 : \text{Int}\rangle\}$$

The value in r_1 should have type "Array($\texttt{sel}(m_0,\ v+1)$)," to reflect the fact that the length of the array is located at address $v+1$ in memory state m_0. We can now use r_2 as the length of array r_1. Even if memory is later changed, for example by updating this record with a new array and different length, we will still be able to use r_2 as the length of r_1 since we remember that they were read from the same memory state m_0.

We generalize the above intuition into the following rule:

$$\frac{\begin{array}{l}\Gamma \vdash e : \text{Ptr } \texttt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle \\ \tau = t_i[{}^{e}\!/_{s}][\texttt{sel}(m,\ e+0)/_{s.0}] \cdots [\texttt{sel}(m,\ e+n-1)/_{s.(n-1)}] \\ \Gamma \vdash m : \texttt{ValidMem}\end{array}}{\Gamma \vdash \texttt{sel}(m,\ e+i) : \tau}\ [\text{Read}]$$

The binding step $\tau = t_i[{}^{e}\!/_{s}][\texttt{sel}(m,\ e+0)/_{s.0}] \cdots [\texttt{sel}(m,\ e+n)/_{s.n}]$ will, for example, convert the field type Array($s.1$) from the previous example to the register type Array ($\texttt{sel}(m,\ v+1)$). The requirement $\Gamma \vdash m : \texttt{ValidMem}$ ensures that we are not in the middle of a dependent update.

Memory updates After writing a value to memory, we must see whether $\Gamma \vdash m : \texttt{ValidMem}$ for the resulting memory state m. If the `store` wrote to a dependent field, then other fields in the record may have to be updated as well in order for the record to be internally consistent once again. For simplicity, our framework requires that all the relevant dependent fields of a record be mutated in the same basic block, with no other intervening writes to the heap. However, it would not be hard to extend the type system to allow invalid memory states that span basic block boundaries.

The rule for stores is below. Starting from a consistent state m, a basic block can perform a series of writes to some object that starts at address e_a. The notation $\texttt{upd}(\cdot, e_a + c_i, e_i)$ represents the result of storing e_i into the object's ${c_i}^{th}$ field; we check that each c_i is in bounds while typechecking the corresponding `store` statement. We ignore duplicate writes to the same field. Regardless of

[1] Throughout this paper we assume that memory is addressed by words, not bytes.

which fields have been overwritten, we can reestablish consistency for this object by checking whether *every* field $e_a + i$ in memory state m' has the type it should. First, we define a function that computes a canonical form for the result of a memory read using standard axioms for memory:

$$\mathrm{Read}(m, e_a + i) = \begin{cases} e & \text{if } m = \mathtt{upd}(m', e_a + i, e) \\ \mathrm{Read}(m', e_a + i) & \text{if } m = \mathtt{upd}(m', e_a + j, e) \text{ and } i \neq j \\ \mathtt{sel}(v_{mem},\ e_a + i) & \text{if } m = v_{mem} \end{cases}$$

With this function we can write the axiom for validating a sequence of writes to the same record:

$$\frac{\begin{array}{l} m' = \mathtt{upd}((\ldots \mathtt{upd}(m, e_a + c_1, e_1)\ldots), e_a + c_j, e_j) \\ \Gamma \vdash e_a : \mathrm{Ptr}\ \mathtt{Rec}_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle \\ \forall 0 \leq i < n\ .\ \Gamma \vdash \mathrm{Read}(m', e_a + i) : \tau_i \\ \qquad \text{where } \tau_i = t_i[{}^{e_a}\!/_{s}][{}^{\mathrm{Read}(m', e_a + 0)}\!/_{s.0}] \ldots [{}^{\mathrm{Read}(m', e_a + n)}\!/_{s.n}] \\ \Gamma \vdash m : \mathtt{ValidMem} \end{array}}{\Gamma \vdash m' : \mathtt{ValidMem}}\ [\text{Update}]$$

For example, consider a record that contains an array reference, its length, and one other field of type Foo. Suppose r_2 contains an array pointer and that r_3 contains its length:

$$\begin{array}{l} \Delta = \{\mathrm{r}_1 = v_{ptr};\ \mathrm{r}_2 = v_2;\ \mathrm{r}_3 = v_3\} \\ \Gamma = \{v_{ptr} \mapsto \mathrm{Ptr}\ \mathtt{Rec}_s.\langle 0 : \mathrm{Array}(s.1);\ 1 : \mathrm{Int};\ 2 : \mathrm{Foo}\rangle; \\ \qquad v_2 \mapsto \mathrm{Array}(v_3);\ v_3 \mapsto \mathit{Int};\ v_{mem0} \mapsto \mathtt{ValidMem}\} \end{array}$$

Now we update the memory state v_{mem0} writing v_2 at address r_1 and v_3 at address r_1+1, therefore mutating both the array and length fields of the record. These two store instructions produce the memory state

$$m' = \mathtt{upd}(\mathtt{upd}(v_{mem0}, v_{ptr}, v_2),\ v_{ptr} + 1,\ v_3)$$

The intermediate memory state $\mathtt{upd}(v_{mem0}, v_{ptr}, v_2)$ is not consistent, and in general it must not be used for `load` instructions. But m' is consistent. Observe that we get

$$\begin{array}{l} \mathrm{Read}(m', v_{ptr} + 0) = v_2 \\ \mathrm{Read}(m', v_{ptr} + 1) = v_3 \\ \mathrm{Read}(m', v_{ptr} + 2) = \mathtt{sel}(v_{mem0},\ v_{ptr} + 2) \end{array}$$

Each of these three fields has the correct type. v_2 has type

$$\begin{array}{rl} \mathrm{Array}(v_3) &= \mathrm{Array}(\mathtt{sel}(m',\ v_{ptr} + 1)) \\ &= \mathrm{Array}(s.1)[{}^{\mathrm{Read}(m', v_{ptr} + 1)}\!/_{s.1}] \end{array}$$

while v_3 has type Int. Location $v_{ptr} + 2$ was not modified, so we rely on the fact that "$\Gamma \vdash v_{mem0} : \mathtt{ValidMem}$" holds to ensure that $\mathtt{sel}(m',\ v_{ptr} + 2) = \mathtt{sel}(v_{mem0},\ v_{ptr} + 2)$ has a value of type Foo.

Checking basic blocks Now we can put these rules together to create a complete algorithm for typechecking a basic block according to the type policy.

The transition function for symbolic evaluation is straightforward. The effect of each instruction on a state $\langle \Delta, \Gamma, m \rangle$ is as follows:[2]

$$\langle \Delta, \Gamma, m \rangle \vdash \texttt{mov}\ \mathrm{r}_{dest}, c \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto c],\ \Gamma,\ m \rangle$$
$$\langle \Delta, \Gamma, m \rangle \vdash \texttt{load}\ \mathrm{r}_{dest}, \mathrm{r}_a \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \texttt{sel}(m,\ \Delta(\mathrm{r}_a))],\ \Gamma,\ m \rangle$$
$$\langle \Delta, \Gamma, m \rangle \vdash \texttt{store}\ \mathrm{r}_{src}, \mathrm{r}_a \Downarrow \langle \Delta,\ \Gamma,\ \texttt{upd}(m, \Delta(\mathrm{r}_a), \Delta(\mathrm{r}_{src})) \rangle$$
$$\langle \Delta, \Gamma, m \rangle \vdash \texttt{add}\ \mathrm{r}_{dest}, \mathrm{r}_{s1}, \mathrm{r}_{s2} \Downarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \Delta(\mathrm{r}_{s1}) + \Delta(\mathrm{r}_{s2})],\ \Gamma,\ m \rangle$$

The other ALU operations have rules similar to `add`. In addition to updating the state, we check that r_a contains a valid pointer in each `load` and `store` operation ($\Gamma \vdash \Delta(\mathrm{r}_a) : \mathrm{Ptr}\ \sigma$).

We assume that each basic block is annotated with an invariant in the form of a typechecker state $\langle \Delta_0, \Gamma_0, m_0 \rangle$, which we use as the initial state of our symbolic evaluation for the block. Evaluation then proceeds according to the transition rules above until we reach the end of the block. At this point we must check that the current state satisfies the invariant that is attached to the successor block(s).

One interesting case here is branches. The branch "$\texttt{beq}\ \mathrm{r}_1, L_j$" at the end of block B_k means that control will jump to block B_j if $\mathrm{r}_1 = 0$, or fall through to B_{k+1} if $\mathrm{r}_1 \neq 0$. A branch may be a dynamic check of some fact that is interesting to the type policy. So each type policy can define an operation $\texttt{Constrain} : \langle \Delta, \Gamma, m \rangle \to e \to \langle \Delta, \Gamma, m \rangle$ that transforms a state to account for any relevant information in a branch condition. For example, suppose we have a state in which r_1 and r_2 hold the same possibly-NULL pointer to σ:

$$\Delta_1 = \{\mathrm{r}_1 = v_1, \mathrm{r}_2 = v_1\}$$
$$\Gamma_1 = \{v_1 \mapsto \mathrm{MaybeNullPtr}\ \sigma\}$$

Then a typical type policy would define

$$\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 = 0) = \langle \{\mathrm{r}_1 = 0, \mathrm{r}_2 = 0\},\ \{\ \},\ m_1 \rangle$$
$$\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 \neq 0) = \langle \{\mathrm{r}_1 = v_1, \mathrm{r}_2 = v_1\},\ \{v_1 \mapsto \mathrm{Ptr}\ \sigma\},\ m_1 \rangle$$

We must check now that $\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 = 0)$ implies the invariant of B_j, and that $\texttt{Constrain}(\langle \Delta_1, \Gamma_1, m_1 \rangle, \mathrm{r}_1 \neq 0)$ implies the invariant of B_{k+1}.

4 Dependent Types for CCured

We have built a prototype checker and inference system for the CCured type system. CCured enforces type safety for legacy C code by classifying pointers according to their usage. Depending on a pointer's classification, or *kind*, CCured changes the pointer to a "fat" pointer structure that stores *metadata* such as array bounds and run-time type information. Figure 4 shows two such fat pointers that we support in our prototype implementation: RTTI pointers, which hold

[2] Changes to the program counter are omitted.

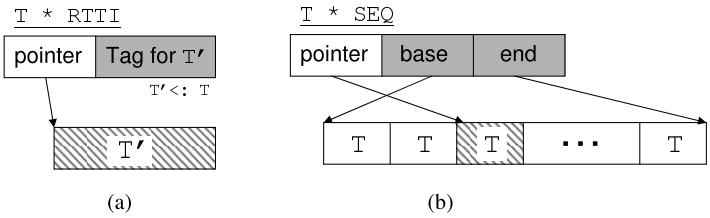

Fig. 4. Two "fat" pointer kinds used by CCured: (a) a pointer with run-time type information, and (b) a sequence pointer (array). The current targets of the pointers are shown with stripes, and the metadata added by the CCured code transformation is in grey.

Run-Time Type Information specifying the dynamic type of the object being pointed to, and Sequence pointers, which are used for arrays. The metadata is used to support run-time checks that CCured inserts when the pointer is dereferenced (for SEQ) or cast (for RTTI). When we want to update a pointer in memory, we may have to update all of the fields in the fat pointer.

4.1 RTTI pointers

Figure 4(a) shows a two-word pointer that refers to a structure in memory and has a type tag specifying the run-time type of the object being pointed to. CCured stores the tag alongside the pointer instead of with the object itself for the sake of a less invasive transformation: the striped location could be in the middle of an array or a struct, and changing its representation to accommodate a type tag would mean transforming all accesses to the base type as well.

RTTI pointers are governed both by a static type (`T` in Figure 4) and the dynamic type specified by the tag (`T′`), which must be a subtype of the static type. Before casting this pointer to a different type `T″`, a program must check that the tag represents a subtype of `T″`. CCured implements these checks using a global table that relates tag values to types.

The assembly-level definition of an RTTI pointer is given in Figure 5. The $\mathrm{Rtti}_\sigma(x)$ type constructor defines a possibly-NULL pointer that has the static type "pointer to σ" but that also has the type denoted by tag x. We use the function `typeof` here to encode the tags-to-types relation for each program.

Our prototype does not yet handle CCured's tagged unions or variable-argument functions, which require reasoning similar to RTTI pointers.

4.2 Sequence pointers

CCured uses Sequence pointers to support arrays and pointer arithmetic in C. A Sequence pointer is a three-word fat pointer, as shown in Figure 4(b), consisting of the actual pointer and pointers to the two ends of the array.

The assembly-level encoding of these pointers is shown in Figure 5. The definition of Seq_σ directly follows the invariants that CCured maintains for its

$$\begin{aligned}
\text{RTTI pointer to } \sigma &= \texttt{Rec}_s.\langle 0 : \mathit{Rtti}_\sigma(s.1);\ 1 : \text{Int}\rangle \\
\text{Sequence pointer to } \sigma &= \texttt{Rec}_s.\langle 0 : \text{Seq}_\sigma(s.1, s.2);\ 1 : \text{Int};\ 2 : \text{Int}\rangle
\end{aligned}$$

where

$$\begin{aligned}
\text{Rtti}_\sigma(t) &\triangleq \{p \mid (p = 0 \vee p \text{ isPtr } \sigma) \wedge (p = 0 \vee p \text{ isPtr } \texttt{typeof}(s.1)) \\
\text{Seq}_\sigma(b, e) &\triangleq \{p \mid (b < e) \ \wedge\ (e - b) \bmod \texttt{sizeof}(\sigma) = 0 \\
&\qquad \wedge (p - b) \bmod \texttt{sizeof}(\sigma) = 0 \\
&\qquad \wedge \forall i.(b \leq (p + i \cdot \texttt{sizeof}(\sigma)) < e) \Rightarrow ((p + i \cdot \texttt{sizeof}(\sigma)) \text{ isPtr } \sigma)\}
\end{aligned}$$

Fig. 5. The meanings of the Rtti and Seq type constructors used by CCured. We use the set comprehension notation $\{x|\ldots\}$ to show the meanings of the types constructors, where "e isPtr σ" means that value e is a pointer to a record with type σ. The $<$ and $\leq$ operators used here are unsigned comparisons.

Sequence pointers: sequence is non-empty and both the end pointer and the actual pointer are aligned on multiples of the element size, although the pointer itself may be out of bounds. We can dereference a Seq_σ pointer p and treat it as an ordinary σ pointer if it is within its bounds b and e. Moreover, we can apply pointer arithmetic to this value, so long as the quantity being added is a multiple of the element size. If the new value is within the bounds, it too can be dereferenced.

To encode Sequence pointers in a type policy for our framework, we define a type constructor $\text{Seq}_\sigma(b, e)$ for each base type σ used by the program. We also define a constructor $\text{CheckedSeq}_\sigma(b, e)$ that represents a sequence pointer after a bounds check:

$$\begin{aligned}
&\text{CheckedSeq}_\sigma\ (b, e) \triangleq \\
&\qquad \{p \mid (b < e) \ \wedge\ (e - b) \bmod \texttt{sizeof}(\sigma) = 0 \ \wedge\ (p - b) \bmod \texttt{sizeof}(\sigma) = 0 \\
&\qquad\quad \wedge \forall i.(b \leq (p + i \cdot \texttt{sizeof}(\sigma)) < e) \Rightarrow ((p + i \cdot \texttt{sizeof}(\sigma)) \text{ isPtr } \sigma) \\
&\qquad\quad \wedge\ b \leq p < e\}
\end{aligned}$$

CheckedSeq_σ has all of the properties of Seq_σ, meaning that we can do pointer arithmetic on it, as well as the property that the current value of the pointer is in bounds and can be dereferenced immediately. In the subtyping relationship used by `IsSubtype` and `TJoin`, $\text{CheckedSeq}_\sigma(e_b, e_e)$ is a subtype of both $\text{Seq}_\sigma(e_b, e_e)$ and Ptr σ.

Whenever our typechecker sees a bounds-checking branch instruction[3] for a value v_p that has type $\text{Seq}_\sigma(e_b, e_e)$, the `Constrain` operation refines the type of v into $\text{CheckedSeq}_\sigma(e_b, e_e)$. Now the value v_p can be dereferenced: the requirement in rules [Read] and [Update] that v_p have a pointer type is satisfied by the rule [Subsumption] and the fact $\texttt{IsSubtype}(\text{CheckedSeq}_\sigma(e_b, e_e),\ \text{Ptr } \sigma)$.

[3] CCured checks both the lower and upper bounds of a sequence pointer in one branch instruction, by using the unsigned comparison $(\mathit{pointer} - \mathit{base}) < (\mathit{end} - \mathit{base})$.

For pointer arithmetic, we can define a type constructor MultipleOf(e) for the integers that are multiples of some value e, and we use the rules

$$\begin{aligned}
&\texttt{ArithType}(\text{Single}(c),\ \times,\ \text{Int}),\ (\text{where } c \text{ is a power of two}[4]) &&= \text{MultipleOf}(c)\\
&\texttt{ArithType}(\text{Seq}_\sigma(e_b, e_e),\ +,\ \text{MultipleOf}(\texttt{sizeof}(\sigma))) &&= \text{Seq}_\sigma(e_b, e_e)\\
&\texttt{ArithType}(\text{CheckedSeq}_\sigma(e_b, e_e), +,\ \text{MultipleOf}(\texttt{sizeof}(\sigma))) &&= \text{Seq}_\sigma(e_b, e_e)
\end{aligned}$$

These rules let us assign the correct type $\text{Seq}_\sigma(e_b, e_e)$ to "$p + 4x$", where p has type $\text{CheckedSeq}_\sigma(e_b, e_e)$ and σ is 4 words long.

4.3 Other features

Besides Rtti and Seq, our type system for CCured uses the basic type constructors you would expect for C code, such as MaybeNullPtr and Int. For each struct or base type defined in the source code, we create a record type σ.

Initialization. Allocation in C programs is done via calls to `malloc` or a related function. It is important to check that the newly-allocated data is initialized correctly. When allocating a record type that contains only non-dependent Ints, no initialization is needed since even garbage values are well-typed. But if the record contains pointer or dependent fields, those fields must be initialized to NULL. (By design, NULL is a valid value for every field type in CCured.)

Stack-allocated data. To support a common C programming idiom, we allow programs to take the address of locations on the stack and pass these pointers to other functions. Typically, this is done to achieve call-by-reference behavior. We require, however, that programs not store such pointers into heap locations or return them from functions. This restriction ensures that when the stack frame is deallocated, there are no dangling pointers into that stack frame. CCured's inference engine can tell us which arguments may be pointers to stack-allocated memory; the verifier needs simply to check that these pointers are not allowed to "escape" through the heap or a return value.

5 Implementation

We have implemented a prototype verifier for the output of CCured using the design in this paper. We use CCured to instrument C programs for type safety, and `gcc 3.3.3` to optimize and compile the code to x86 assembly. Our verifier uses abstract interpretation over the domain of symbolic expressions to infer register types and ensure that every instruction preserves memory safety. Our implementation can handle Sequence and RTTI pointers and their associated dependencies. We also implement pointers to stack-allocated data.

The CCured code transformer will generate annotations for each program that serve as a partial witness of the program's correctness, but these annotations need not be trusted. Incorrect annotations will result in failed verification

[4] When c is not a power of two, we need a branch instruction to check the result of the multiplication for overflow before assuming that the product is a multiple of c.

rather than unsoundness, just as incorrect type information in Java bytecode will result in failed typechecking. The annotations encode: (1) the type of every global variable; (2) the global table of RTTI tags; (3) for each function, the types of its arguments and return value, and the types and stack location of any local variables that will have their address taken; and (4) for every call to `malloc`, the type that will be applied to the resulting pointer (e.g. "`T*`" if the source instruction is "`T* var = (T*) malloc(`e`)`"). Annotations are expressed in inline assembly so that GCC will pass them from the instrumented source code down to the verifier. Only the annotations for `malloc` appear in the middle of a function, ensuring that this inline assembly will impose minimal constraints on the optimizer. Other annotation strategies would also be feasible.

These annotations give us all the information we need to know about the structure of the heap. All that remains is to infer types for registers and check each instruction for memory safety.

Considerable engineering work needs to be done before our verifier will be able to support all of the features of C. The prototype does not yet support variable-argument functions, tagged unions, floating point operations, or function pointers. We have not implemented any fat pointer kinds other than Sequence and RTTI, although most other kinds (such as "forward-sequence" and kinds that combine RTTI with bounds information) will be straightforward. We also do not support casts between Sequence pointer types that have different base types. Such casts are rare, and we may need CCured to annotate them so that they can be verified.

In order to facilitate joins, our abstract interpreter limits the form of symbolic expressions that are used for pointer arithmetic. Pointer offsets may be either constants or multiples of the base type size. This works well for one-dimensional arrays, but not for nested arrays. We are currently examining how to support more general indexing expressions without losing precision in our join algorithm.

We treat calls to `malloc` and other allocation functions specially, and deal with initialization as described in Section 4.3. CCured uses the Boehm-Demers-Weiser garbage collector [11], which we trust, so calling `free` has no effect.

5.1 Experiments

As an initial test, we used our prototype on the `go` program in the Spec95 benchmark suite. Of the Spec95 programs, we chose `go` because it makes extensive use of arrays while avoiding floating-point instructions, which our x86 parser does not yet handle. We used the `-O2` optimizer flag while compiling the program.

Of the 378 functions in the 29,321 LOC program, we can successfully verify 316 of them(84%). The most common reason for failure was that array indexing expressions of nested arrays are too complicated for our abstract domain. We directed CCured to flatten two-dimensional arrays into single-dimensional arrays, but in general there is no way to do this for arrays of structs that themselves contain arrays. Other failures were due to the unimplemented C features mentioned earlier. We are currently working to improve the implementation so that we can verify all of the Spec95 suite.

Verifying the program takes 194 seconds on a 2.4 GHz Pentium 4 with 1 GB of RAM. While testing our system, we discovered several soundness bugs in CCured: the instrumentation did not safely handle NULL return values from `malloc`, and CCured's optimizer incorrectly removed bounds checks based on the faulty assumption that two pointers couldn't alias. This experience shows the importance of independent verification of safety tools.

6 Related Work

Certified object code. There has been much work done to certify that binary code adheres to various safety properties. Colby et al. [12] survey several approaches, such as TAL and PCC, and describe the general problem of certifying mobile code, including how such certifications can be communicated to the end user.

Typed Assembly Language [8,4] is used as a compilation target for Popcorn, a subset of C. TAL includes many useful features, including flow-sensitive types for registers so that register types can change from one instruction to the next; typechecking that is done one basic block at a time; existential types; and support for stack-based compilation schemes [9]. But TAL does not support the dependent types that we need for CCured, and it assumes that assembly code is generated by a specially-written, type-preserving compiler.

Proof-Carrying Code [13,14] packages object code with a checkable proof of safety. The original implementations of PCC targeted specific type policies, such as Java's type system [14]. Recent projects such as LTT [15] and work by Shao et al. [16] seek a general type system for certified code that is not tied to any one source language. A low-level type system permits use of a wide variety of proofs and proof techniques, and it allows code from multiple source languages to be combined safely. But these two systems do not yet target imperative languages, making them impractical for the applications we are considering.

Producing checkable proofs is a goal for our type system as well. Our approach will follow work done by the Open Verifier group to design an extensible system for foundational verification [17]. Currently, our implementation uses the Open Verifier's code for checking that stacks and function calls are handled correctly.

Balakrishnan and Reps [18] present a system for analyzing memory accesses in x86 code. They do not require annotations from the compiler, but in exchange they trade off some precision and soundness.

Dependent types. The Xanadu language [19] provides an expressive dependent type system for an imperative, source-level language. Xanadu supports dependencies between different objects, which lets the language express more interesting properties about heap structures than ours can.

Xanadu can be compiled to DTAL, a dependently-typed assembly language [5]. DTAL focuses largely on array types and array-bound check elimination. Basic blocks are annotated with invariants to reduce the need for type inference, and a type-preserving compiler is used. DTAL does not support modification of dependently-typed locations in the heap.

Our restricted form of dependent types is similar to Hickey's very dependent function types [7]. Hickey encodes immutable records as functions from labels

to values. By using very dependent types for these functions, one can impose dependencies among the object's fields. Hickey uses these types to formalize a theory of objects, including methods and inheritance. Our type system has a similar focus on dependencies among fields and function arguments, but in the context of a low-level imperative language with mutable structures.

Grossman [20] discusses the difficulty in supporting destructive updates in a language with existential types; this is the same difficulty that our system addresses for dependent types.

7 Discussion

We have described a dependently typed assembly language that supports destructive updates of dependent values that are stored in the heap. We can express in this framework the invariants enforced by CCured in the instrumented programs it outputs, and we can check statically that they are maintained. Our prototype verifier for CCured demonstrates that our approach can be used in practice.

Future work on this project will proceed in three main directions. First, we will apply our framework to type policies other than CCured. Already we have created an extension for Cqual [2], an interprocedural static analysis tool that infers type qualifiers for C programs and has been used to check several important security properties [21,22].

The second direction is to generalize our system of dependent types. Our types work well for dependencies between two local variables or two fields of the same object, but they cannot encode dependencies between two memory locations that are not stored in the same object. Removing this limitation will allow us to encode all or nearly all invariants of the source languages we are dealing with, including downcasts in object-oriented code and null-terminated strings.

The third direction of future work is to produce a proof of type safety for each program. Currently, the verifier and type policy are treated as part of the trusted computing base. Through the Open Verifier project [10], we plan to produce "foundational" proofs that can be checked by end users who would not need to trust our type inference or the implementation of the type policy.

References

1. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems **27** (2005)
2. Foster, J.S., Terauchi, T., Aiken, A.: Flow-Sensitive Type Qualifiers. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany (2002) 1–12
3. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. 7th USENIX Security Conference, San Antonio, Texas (1998) 63–78

4. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems (**21**)
5. Xi, H., Harper, R.: Dependently Typed Assembly Language. (In: The Sixth ACM SIGPLAN Int'l Conference on Functional Programming)
6. Chang, B.Y.E., Chlipala, A., Necula, G., Schneck, R.: Type-based verification of assembly language for compiler debugging. In: The 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation. (2005) 91–102
7. Hickey, J.: Formal objects in type theory using very dependent types. In: Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages. (1996)
8. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A realistic typed assembly language. In: Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software. (1999) 25–35
9. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. In: Proceedings of the Second International Workshop on Types in Compilation, Springer-Verlag (1998) 28–52
10. Schneck, R.R.: Extensible Untrusted Code Verification. PhD thesis, University of California, Berkeley (2004)
11. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software—Practice and Experience (1988) 807–820
12. Colby, C., Crary, K., Harper, R., Lee, P., Pfenning, F.: Automated techniques for provably safe mobile code. Theor. Comput. Sci. **290** (2003) 1175–1199
13. Necula, G.C.: Proof-carrying code. In: The 24th Annual ACM Symposium on Principles of Programming Languages, ACM (1997) 106–119
14. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A certifying compiler for java. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, ACM Press (2000) 95–107
15. Crary, K., Vanderwaart, J.C.: An expressive, scalable type theory for certified code. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, ACM Press (2002) 191–205
16. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 217–232
17. Chang, B.Y.E., Chlipala, A., Necula, G.C., Schneck, R.R.: The Open Verifier framework for foundational verifiers. In: The 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation. (2005) 1–12
18. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 binary executables. In: Proc. Compiler Construction (LNCS 2985), Springer Verlag (2004) 5–23
19. Xi, H.: Imperative programming with dependent types. In: Proceedings of 15th IEEE Symposium on Logic in Computer Science, Santa Barbara (2000) 375–387
20. Grossman, D.: Existential types for imperative languages. In: Proceedings of the 11th European Symposium on Programming Languages and Systems. (2002) 21–35
21. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting Format String Vulnerabilities with Type Qualifiers. In: Proceedings of the 10th Usenix Security Symposium, Washington, D.C. (2001)
22. Johnson, R., Wagner, D.: Finding user/kernel pointer bugs with type inference. In: Proceedings of the 13th USENIX Security Symposium. (2004)