

Formalizing the SAFECode Type System

Daniel Huang and Greg Morrisett

Harvard University, Cambridge MA 02138, USA

`dehuang@fas.harvard.edu`,

`greg@eecs.harvard.edu`

Abstract. The Secure Virtual Architecture (SVA) provides an object-level integrity policy, similar to type-safety, for languages such as C and C++, and thus rules out a wide range of common vulnerabilities. SVA uses an enhanced version of the Low-Level Virtual Machine (LLVM) compiler called SAFECode to enforce the policy through a combination of static and dynamic type-checks. However, this results in a relatively large trusted computing base (TCB). SVA reduces the TCB with an unverified type-checker that relies upon a paper-and-pencil proof of type-soundness for a core-language. As a further step towards increasing the assurance of the compiler, we present a mechanized proof of soundness and a verified type-checker for a realistic subset of the SAFECode type system developed using the Coq Proof Assistant.

Keywords: verification, SAFECode, LLVM, memory safety.

1 Introduction

Most of our computing infrastructure is coded using low-level languages such as C/C++. Unsurprisingly, it is easy to make simple mistakes in these languages that lead to well-known vulnerabilities. In principle, recoding the infrastructure in a type-safe language would eliminate many of these vulnerabilities, but the costs of doing so seem to outweigh the benefits.

An attractive alternative is to bring the benefits of type-safety to legacy code by combining static analyses and run-time checks to automatically enforce a type-safety policy. There are many challenges in doing this effectively, as static analyses are generally too weak to reason effectively about real C/C++ programs, resulting in many false positives. The cost of inserting run-time checks and maintaining the meta-data needed to support those checks can also be prohibitively expensive. Recently, a number of systems have successfully combined the benefits of static analysis, dynamic checks, program optimization, and clever run-time representations to produce viable solutions [3,10,11,5,4].

One such system, SAFECode [5], uses sophisticated analyses and optimizations to eliminate run-time checks. However, this adds the SAFECode compiler to the trusted computing base (TCB). We could try to prove that the analyses and transformations (and subsequent optimizations) are correct as in the CompCert project [8]. Perhaps more easily, we can build a verified checker that

<i>Type</i>	$\tau ::= \text{int} \mid \text{char} \mid \text{unknown} \mid \tau * \rho \mid \text{handle}(\rho, \tau)$
<i>Statements</i>	$S ::= \epsilon \mid S; S \mid x = E \mid \text{store } E, E \mid \text{storeToU } x, E, E$ $\mid \text{storec } E, E \mid \text{storecToU } E, E \mid \text{poolfree}(E, E)$ $\mid \text{poolinit}(\rho, \tau)x\{S\} \mid \text{pool}\{S\}\text{pop}(\rho)$
<i>Expressions</i>	$E ::= \text{var} \mid V \mid E \text{ op } E \mid \text{load } E \mid \text{loadFromU } x, E \mid \text{loadc } E$ $\mid \text{loadcFromU } E \mid \text{cast } E \text{ to } \tau \mid \text{poolalloc}(x, E)$ $\mid (x, \&E[E]) \mid \text{castint2pointer } x, E \text{ to } \tau$
<i>Value</i>	$V ::= \text{uninit} \mid \text{Int} \mid \text{region}(\rho)$

Fig. 1. Core-language presented in original SAFECODE paper

attempts to prove that rewritten and optimized code respects the SAFECODE security policy. The goal of this paper is to increase the assurance of the SAFECODE compiler by formalizing a realistic subset of the language and its type system, presenting a mechanically-checked proof of soundness, and building a verified checker that can be used to check code emitted by SAFECODE.

2 Overview of SAFECODE

Our work builds upon a previous paper describing the SAFECODE system [5], which enforces an object-level integrity policy similar to, but weaker than traditional notions of type-safety. Conceptually, SAFECODE instruments all dangerous operations such as loads and stores with dynamic checks. To justify the elimination of unnecessary run-time checks, the paper formalized a core-language, type system and gave a paper-pencil proof of soundness for the typing rules.

We have reproduced their core-language in Figure 1. SAFECODE uses regions, similar to the approach pioneered by Tofte and Talpin [12] and later refined in Cyclone [6]. Like these previous systems, a pointer type $\tau * \rho$ is indexed by a region variable ρ indicating the region of memory it references. However, SAFECODE only places objects of the same type in a given region, allowing region metadata to support efficient run-time casts. Objects whose type cannot be statically determined are put in untyped regions. The type system tracks which regions are accessible, and hence, which pointers can be safely dereferenced.

More concretely, SAFECODE provides a lexically scoped construct of the form `poolinit(ρ , τ) $\{\dots\}$` , which allocates a new region (or pool) to exclusively hold values of type τ and binds the region to ρ . Regions are typically represented as lists of pages that can be dynamically grown as new objects are allocated. Initially, the pages are zero-filled and zero is assumed to be a valid value for any type. In particular, dereferencing address zero will result in a trapped error (segmentation fault). Within the scope of the `poolinit`, programs can allocate objects of type τ in ρ using `poolalloc`, which returns a pointer of type $\tau * \rho$. Such pointers can be dereferenced (via `load`), updated (via `store`), or used to deallocate the object (via `poolfree`) while in the scope of ρ . Memory reclaimed in a region can be recycled for use at the same type. At the end of `poolinit`'s