Integrating Types and Specifications for Secure Software Development

Greg Morrisett

Harvard University,
Cambridge, Massachussetts, 02138, USA
greg@eecs.harvard.edu
http://www.eecs.harvard.edu/~greg

Abstract. Today, the majority of security errors in software systems are due to implementation errors, as opposed to flaws in fundamental algorithms (e.g., cryptography). Type-safe languages, such as Java, help rule out a class of these errors, such as code-injection through buffer overruns. But attackers simply shift to implementation flaws above the level of the primitive operations of the language (e.g., SQL-injection attacks). Thus, next-generation languages need type systems that can express and enforce application-specific security policies.

Keywords: dependent types, verification, software-security.

1 Overview

> *In theory, there is no difference between theory and practice. But, in practice, there is.*
>
> Jan L. A. van de Snepscheut

Most security problems today are rooted in implementation errors: failure to check that an array index stays in bounds, failure to check that an input string lacks escape characters, failure to check that an integer passed to an allocation routine is positive, *etc.* Furthermore, the techniques we use for validating that code is free from these errors (fuzz testing, manual inspection, static analysis tools, *etc.*) have proven woefully inadequate. For example, in spite of a large security push starting in 2002, hackers are still finding buffer overruns in Microsoft's operating system and other applications.

The irony is that many of the simplest kinds of errors, such as buffer overruns, could be prevented by the use of a type-safe language instead of C or C++. This is because a type-safe language is required to enforce the basic abstractions of the language through a combination of static and dynamic tests. Languages such as Java, Scheme, and ML are all examples of languages where, at least in principle, buffer overruns cannot occur.

I. Kotenko and V. Skormin (Eds.): MMM-ACNS 2010, LNCS 6258, pp. 32–35, 2010.

However, in practice there are four problems with today's type-safe languages:

1. It is expensive to re-write programs in new languages. For example, Windows consists of more than 60 million lines of code.
2. Today's type-safe languages perform poorly when compared to C or C++, particularly for systems-related tasks (e.g., operating systems, networking, databases, etc.)
3. Today's languages check for buffer overruns at run-time and throw an exception that is rarely caught. This shifts the flaw from a possible code injection to a denial of service attack.
4. The type systems for today's languages are too weak to enforce policies needed to stop next-generation attacks.

The first problem is a key issue for legacy systems, but not for next-generation environments (*e.g.*, cell phones, tablets, *etc.*) Furthermore, for key segments of the software market, notably the medical, military and financial industries, the cost of developing highly secure, new software is practical.

The other three problems require fundamental new research in the design of systems programming languages. On the one hand, we need a way to express rich, application-specific security policies and automatically check that the code respects those policies. On the other hand, we need languages that, like C and C++, provide relatively direct access to the underlying machine for performance-critical code.

2 Refinement Types

A number of researchers are looking at next-generation programming languages that support *refinement types*. Refinement types take the form "$\{x : T \mid P(x)\}$" where T is a type and P is a predicate over values of the type T. For example, the type $\{x : \texttt{int} \mid x > 0\}$ captures the set of all positive integers.

The principal challenge with refinement types is finding a way to support type-checking. Some languages, such as PLT Scheme [1], rely upon dynamic checks, so that when a value is "cast" to have a refinement type, the predicate is evaluated on the value, and if it fails, an exception is thrown. This is a simple and expedient way to incorporate refinements, but leads to a number of problems.

First, it restricts the language of predicates that we can use to a decidable fragment. Second, the semantics of these run-time checks are not clear, especially when the predicates can have side effects or when the predicates involve mutable data shared amongst threads. Third, as noted with array-bounds checks, the potential for dynamic failure (an exception) provides for a possible denial of service attack.

To address this last problem, many systems, such as JML# [2] try to discharge these tests at compile time using an SMT theorem prover. In practice, this works well for simple predicates (e.g., linear constraints on integers), but less well for "deep predicates" (e.g., this string is well-formed with respect to this grammar.) Furthermore, SMT provers are focused on fragments of first-order logic (with

particular theories). In practice, we have found that, just as programs need to abstract over other sub-programs, specifications need support for abstractions, and higher-order logics provide a powerful way to achieve this.

3 Type-Theoretic Refinement

Proof assistants, such as Coq [3], provide a powerful, uniform way to write (a) programs, (b) specifications and models that capture desired properties of programs ranging from simple typing and safety properties up to full correctness, and (c) formal, machine-checked proofs that a given program meets its specification. They sacrifice automation for finding proofs that code is well-formed, relying instead upon programmers to explicitly construct these proofs. In this sense, they are less convenient than fully automated type-checking techniques. But they are far less limited than the approaches listed above.

For example, Xavier Leroy and his students have used Coq to construct an optimizing compiler that translates a (well-defined) subset of C to PowerPC code, defined operational semantics for both C and PowerPC code, and mechanically proved that when the compiler succeeds in producing target code, that code behaves the same as the source code, thereby establishing the correctness of the compiler [4]. Coq is not alone in providing support for this style of program development: Other examples include ACL2 [5], Agda [6], Epigram [7], and Isabelle [8].

Nevertheless, today's proof assistants suffer from a number of limitations that limit their applicability. One serious shortcoming is that we are limited to writing and reasoning about only purely functional programs with no side effects, including diverging programs, mutable state, exceptions, I/O, concurrency, etc. While some programming tasks, such as a compiler, can be formulated as a pure, terminating function, most cannot. Furthermore, even programs such as a compiler need to use asymptotically efficient algorithms and data structures (*e.g.*, hash-tables) but current dependently typed languages prevent us from doing so. Thus a fundamental challenge is scaling the programming environments of proof assistants to full-fledged programming languages.

4 Ynot

For the past few years, my research group has been investigating a design for a next-generation programming language that builds upon the foundation provided by proof-assistants. We believe that environments, such as Coq, that provide powerful tools for specification and abstraction provide the best basis moving forward, and thus the central issues are (a) how to incorporate support for computational effects, and (b) how to scale proof development and maintenance to real systems.

In the case of effects, we developed a modest extension to Coq called Ynot, which is based on Hoare Type Theory (HTT) [9]. HTT makes a strong distinction between types of pure expressions, and those that may have side-effects, similar

to the modality found in Haskell's monadic treatment of IO and state. Impure expressions are *delayed*, and their effects only take place when they are explicitly run. Because impure expressions are delayed, they can be treated as "pure" values, avoiding some of the problems with refinements in the presence of effects.

In addition to extending Coq with support for effects, the Ynot project has investigated techniques for effective systems programming. For example, we built a small, relational database management system using Ynot which was described in previous work [10]. This included an optimizing query compiler, as well as complicated, pointer-based data structures including hash-tables and B+-trees. The whole development, including the parser for queries, the query optimizer, the data structures, and execution engine are verified for partial correctness.

Our experience building verified systems software in this fashion is promising, but a number of hard issues remain to be explored. First and foremost, constructing proofs of correctness demands a clean specification for the problem domain. And of course, a bug in the specification can lead to a bug in the code. So one challenge is finding specifications for systems that can be verified in their own right. Another issue is the cost of developing and maintaining proofs. Originally, we coded proofs by hand. Since then, we have shifted towards a semi-automated style that makes liberal use of custom tactics [11]. The latter approach not only cuts the size of the proofs, but makes them far more robust to changes in the program or specification.

Finally, the programming language embedded in Coq is a relatively high-level, ML like language. For many applications, it is ideal, but for many systems programming tasks (e.g., hypervisors or device drivers), it is too high-level. Thus, we still lack a good low-level programming environment which can effectively replace C.

References

1. PLT Scheme, `http://www.plt-scheme.org/`
2. Leavens, G.T., et al.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)
3. The Coq Proof Assistant, `http://coq.inria.fr/`
4. Leroy, X.: Formal verification of a realistic compiler. Comm. of the ACM. 52(7), 107–115 (2009)
5. ACL2, `http://userweb.cs.utexas.edu/~moore/acl2/acl2-doc.html`
6. Agda, `http://www.cs.chalmers.se/~catarina/agda/`
7. Epigram, `http://www.e-pig.org/`
8. Isabelle, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
9. Nanevski, A., et al.: Polymorphism and separation in Hoare Type Theory. In: 11th ACM Intl. Conf. on Functional Prog, pp. 62–73. ACM Press, New York (2006)
10. Malecha, G., et al.: Towards a verified relational database management system. In: 37th ACM Symp. on Principles of Prog, pp. 237–248. ACM Press, New York (2010)
11. Chlipala, A., et al.: Effective interactive proofs for higher-order imperative programs. In: 14th ACM Intl. Conf. on Functional Prog, pp. 79–90. ACM Press, New York (2009)