J Inf Process Syst, Vol.14, No.2, pp.309~321, April 2018
https://doi.org/10.3745/JIPS.01.0027

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Verifying Code toward Trustworthy Software

Hyong-Soon Kim* and Eunyoung Lee**

Abstract
In the conventional computing environment, users use only a small number of software systems intensively. So it had been enough to check and guarantee the functional correctness and safety of a small number of giant systems in order to protect the user systems and their information inside the systems from outside attacks. However, checking the correctness and safety of giant systems is not enough anymore, since users are using various software systems or web services provided by unskilled developers. To prove or guarantee the safety of software system, a lot of research has been conducted in diverse areas of computer science. We will discuss the on-going approaches for guaranteeing or verifying the safety of software systems in this paper. We also discuss the future research challenge which must be solved with better solutions in the near future.

Keywords
Certified Compiler, Formal Verification, Language Semantics, Program Verification

1. Introduction

Due to the popularity of small computing devices such as smartphones, users are surrounded by diverse software more than ever. Software on small computing devices usually shows the characteristics of small size, low power consumption and smaller number of functions. Compared to these lightweight software systems, the conventional software systems show the characteristics of bigger size and more complex functionality. In the conventional computing environment, users use only a small number of software systems intensively. So it had been enough to check and guarantee the functional correctness and safety of a small number of giant systems in order to protect the user systems and their information inside the systems from outside attacks.

However, checking the correctness and safety of giant systems is not enough anymore, since users are using various software systems or web services provided by unskilled developers. To prove or guarantee the safety of software system, a lot of research has been conducted in diverse areas of computer science. Testing or behavior monitoring has been one of the best approaches for insuring the safety and correctness of software system for a long time. Those approaches take a large amount of time and money for verification: paying time and money was worthy since the verified software was used for a long time, and the developers were rich enough to pay the verification cost.

In the era of small software applications with relatively short lifetime, verifying the already-

Manuscript received November 11, 2017; first revision December 26, 2017; accepted December 29, 2017.
Corresponding Author: Eunyoung Lee (elee@dongduk.ac.kr)
* Dept. of ICT Platform & Services, National Information Society Agency, Daegu, Korea (khs@nia.or.kr)
**Dept. of Computer Science & Engineering, Dongduk Women's University, Seoul, Korea (elee@dongduk.ac.kr)

implemented software is not a viable option anymore. Verifying the code in the middle of development or guiding developers to write a safer code would be a more feasible solution. If the language semantics could guarantee that the compiled code is always safe with a rigid proof, software developers would not need time-consuming testing or runtime monitoring any more.

Due to the manifold definition of security, there exists an argument about the meaning of secure software. What does secure software or a secure software system mean? Is the software made by the development process which guarantees the security? Is the software going to be used for guaranteeing the security of users? Or will the software not undermine the security of the underlying system or the user information inside?

Depending on which definition of secure software is used, the approaches toward building secure software will vary. In this paper, the term *trustworthy software* is used to refer a software system which will not do the harmful things if compiled.

We will discuss the on-going approaches for guaranteeing or verifying the safety of software systems in this paper. Fig. 1 shows the research topics and mechanisms of software verification we will examine in this paper. In Section 2, current software verification approaches are categorized by their checking targets, target platforms, type of languages. From Section 3 to Section 5, we will discuss the state-of-the-art approaches of trustworthy software.

In Section 6, the examples of trustworthy software research will be reviewed to demonstrate the viability of trustworthy software development. We also discuss the future research challenge which must be solved with better solutions in the near future. We will conclude our review in Section 7.

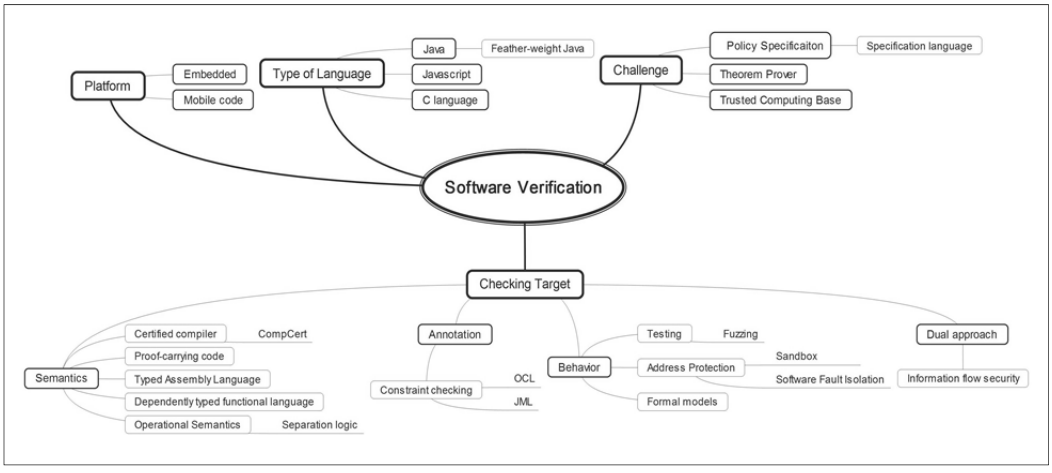

Fig. 1. Taxonomy of software verification research.

2. Categories of Software Verification

Categorizing the methods of software verification can be very tricky since so many algorithms, platforms, or frameworks have been proposed for the exactly same purpose: verifying software in order to guarantee its safe or not-hostile behavior. From the view point of programming language, one simple criterion is whether or not the language semantics is concerned for the resulting code safety.

If the safety of implemented software is checked by the language semantics, it is not necessary to verify the resulting software after development is done. That means, if the code compilation is successful, the code will not do any harm to the system. With these approaches, verification is usually done when writing code and compiling it. Programming languages in this category are usually equipped with fully-verified operational semantics, and dedicated certifying compiler produces runnable machine code and a proof of its safety. Verification of the software safety ends up with checking the validity of the accompanying proof. The operational semantics of these languages and the proofs are usually written in mathematical logic. The proofs can be written by human experts, but it can be a tedious job for a human expert to write down long logical formulas for herself. Producing a proof for a snippet of code and verifying the proof automatically is one of the most interesting issues in semantics-based software verification, and software systems for this purpose are called theorem provers. We will discuss the research trend of theorem provers in Section 6.2.

However, most of the programming languages which are used for software implementation do not have the safety-sensitive operational semantics. In order to overcome the weakness of conventional programming languages such as C or to re-enforce the safe feature of conventional programming languages such as Java, constraint specification languages have been proposed. Constraint specification languages are used for specifying the pre- and post-conditions of functional procedures written in conventional programming language. Constraint are dropped or transformed into a bunch of conditional branch code by the dedicated pre-processors. In the runtime, constraints defined by developers are checked to see whether all the required constraints remain valid. Even though adding annotation is weaker than writing a program in a programming language with safety-sensitive semantics, adding annotation can be a good option when implementing all the software system is already done. We will discuss the constraint specification languages in Section 4.

If the source code is not available for annotation, the annotating approach cannot be applied. In this case, it is inevitable to determine the safety of a software system based on its observable behavior. Software testing and address space protection are two of the most popular software verification techniques which can be used without available source code. We will discuss these techniques in Section 5.

Some security features are so important that some approaches cross our classification boundaries have been proposed. These are: memory space protection and information flow security. Keeping a not-verified application within a limited memory space prevents the untrusted application from underlying system infrastructure or other applications. Sandboxing is monitoring the addressing violation in order to protect the memory address boundary of every application. Since it is a runtime monitoring approach, sandboxing can be applied to software with no source code. Software fault isolation is another approach to protect address boundaries. In this scheme, annotations checking address violation are inserted into the source code of application automatically before compiling. Recently the researchers have started checking the memory safety by language semantics and checkable proofs. Separation logic is the most up-and-coming research in this area. With separation logic, more rigid and tight reasoning can be achieved in protecting memory address spaces. We will discuss sandboxing in Section 5.2, software fault isolation in Section 4.1, and separation logic in Section 3.4.

Information flow security is another good example to which diverse approaches are applied. Information breach of program variables can be successfully handled by program annotation, whereas covert channels such as timing or power consumption can be detected by runtime monitoring. We will discuss the approaches for information flow security in Section 4.2 and Section 5.3. Table 1 shows the

classification of software verification we proposed in this paper.

After extensive overview of software verification mechanisms, we will discuss the research topics which must be explored for the success of software verification. We will present the related topics with some pioneering research examples in Section 6.

Table 1. Classification features of software verification

	PCC	TAL	Dependent type	Separation logic	SFI	Information flow	Fuzzing	Sandbox	PBRT
Considering semantics	Yes	Yes	Yes	Yes	No	No	No	No	Yes
Observing behavior	No	No	No	No	No	Yes	Yes	Yes	Yes
Need of source code	Yes	Yes	Yes	Yes	Yes	Yes/No	Yes/No	No	Yes

3. Checking by Language Semantics

Infusing safety feature into language semantics can be very useful to build safe and trustworthy software. Researches in this area usually span in two ways: formalizing the semantics of (a subset of) existing language or designing a new language with formalized language semantics.

3.1 Proof-Carrying Code

Necula and his colleague [1,2] proposed proof-carrying code (PCC) in which a host system can execute a code from untrusted outside party. In this mechanism, the host system must determine with certainty that it is safe to execute a program supplied (possibly in binary form) by an untrusted (and maybe malicious) outsider. The untrusted code producer must supply with the code a safety proof that claims that the code satisfies a previously defined safety policy.

Appel and his colleague [3,4] proposed foundational proof-carrying code (FPCC), in which code is verified with the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system. The paper describes the mathematical and engineering problems to be solved and how to applying PCC mechanism to general programs.

Recent research of Vanegue [5] made comparison between PCC [1] and FPCC [3,4]. While in PCC, it is possible to make use of type rules directly in the axioms of the system, in FPCC, each type rule should be first defined from ground axioms. In addition, FPCC suggests the use of type-preserving compilers, such as the one in CompCert [6]. Vanague [5] argued that PCC is more vulnerable to unconventional machine instructions than FPCC since PCC can lose soundness because of its dependency to the underlying arbitrary type system.

3.2 Typed Assembly Language

Morrisett and his colleagues [7-9] proposed typed assembly language (TAL) which has been based on other machine code verification. The typed assembly language is based on a conventional RISC assembly language, but its static type system even supports high-level language abstractions, such as closures, tuples, and user-defined abstract data types. The type system ensures that well-typed programs cannot violate these abstractions, and the safety of underlying system.

The TAL has played an important role in verifying low-level system software [10]. Lattner and Adve [11] proposed a compiler framework called LLVM (Low Level Virtual Machine). The framework was designed to support transparent program analysis and transformation for arbitrary programs. It provides high-level information to compiler transformations at compile-time, link-time, and runtime.

Patrignani et al. [12] suggested a compilation technique which prohibits an attacker operating at the target language level from bypassing security features of the source language. In their compilation scheme, whose source language is an object-oriented, high-level language and whose target language is an extended typed assembly language.

3.3 Dependently Typed Functional Language

In type theory, a dependent type is a type whose definition depends on a value. In intuitionistic type theory, dependent types are used to encode logic's quantifiers like "for all" and "there exists" in intuitionistic type theory. Dependent types can enhance the expressive power of type systems, and the improved type system is powerful enough to prevent software bugs [13]. Several functional programming languages support dependent types such as Agda [14], Cayenne [15], Coq [16,17], Epigram [18], and Idris [19].

Due to its expressive power, dependently typed language is utilized in several areas. For example, Jeffrey [20] proposed a compiler back-end and library for web client application development in Agda. The target language of the compiler back-end is JavaScript, so it can be plugged in a web browser.

Coq is one of the most popular proof assistant based on dependently typed functional language. Huet and Herbelin [21] summarized the research related to Coq over 30 year. Researchers in diverse research areas are adopting Coq to verify formally their implementation. Athalye [22] suggested a framework based on Coq, which can be used for formalizing the theory of IO automata, including refinement, simulation relations, and composition. Chatzikyriakidis and Luo [23] utilized Coq for formalizing natural language inference based on the formal semantics in modern type theories (MTTs).

3.4 Separation Logic

It is commonly accepted that verifying a program written in a programming language allowing memory address manipulation such as C is extremely hard or even impossible. However, researchers have demonstrated that carefully designed logic or type system can reason about the safety of program with pointers. Separation logic by Reynolds [24-26] aims at expediting programs that manipulate pointer data structures. The separation logic is used for specifications and proofs of a program component, and it reasons about only the portion of memory used by the component, and not the entire global state of the system.

For checking the proofs of a program, Berdine et al. [27] proposed an automatic proof checker of separation logic. In the proposed separation logic checker, called *Smallfoot*, the assertions describe only the shapes of data structures rather than their detailed contents.

Separation logic has gained widespread popularity because of its ability to succinctly express complex invariants of program's heap configurations. Qui and his colleagues [28,29] proposed a framework which extends the Vcc framework [30] to provide an automated deductive framework against separation logic specifications for C programs based on natural proofs.

4. Annotation and Transformation

Implementing a software system in a programming language with rigorous semantics is the best way of building trustworthy software. However, there still exist a large number of legacy systems written in not-so-much safe programming languages. For some mysterious reasons, a lot of software systems are implemented in unsafe programming languages. To enhance the trustworthiness, adding annotation to the code of classic programming language is a simple and viable option. Usually annotated code is transformed into a code of original programming language by a dedicated pre-processor.

4.1 Software Fault Isolation

It has been believed that protecting memory space or memory pages is the duty of the OS/kernel or the hardware: virtual memory mechanism of hardware or user/admin mode of operating system [31].

However, memory boundaries can be preserved by program annotation. Wahbe et al. [32] proposed a memory protection based on program rewriting called software fault isolation (SFI). In SFI scheme, untrusted object code is modified by rewriting algorithm automatically in order for the untrusted code not to write or jump to an address outside its fault domain.

Recently, SFI scheme has expanded to various intermediate languages or hardware platforms. While conventional SFI relies on analysis of assembly-level programs, Kroll et al. [33] suggested an improved scheme of analyzing and rewriting programs in a compiler intermediate language, the Cminor language of the CompCert C compiler. Since the CompCert compiler has been formally certified that its transformation process preserves the safety property of a program, it is formally guaranteed that resulting binary modules satisfy the SFI memory safety. At the same time, resulting binary modules can be any of the supported architectures of CompCert compiler.

4.2 Information Flow Security

Confidentiality is one of the most important factors of standard system security. An end-to-end confidentiality policy might assert that secret input data cannot be inferred by an attacker when the attacker can observe the system output. To enforce this information flow policy, access control and encryption have been conventionally used. In addition to these conventional protections, the use of programming language techniques for specifying and enforcing information flow policies has been proposed. Sabelfeld and Myers [34] have surveyed language-based approaches for information flow security and identified open challenges in this area of software verification.

Costanzo et al. [35] designed and implemented a system which can be used to formally verify that the end-to-end behavior of the computing system really satisfies various information-flow policies. The input of their system includes a software system that consists of both C and assembly programs, and they demonstrate that their system constructed an end-to-end security proof, fully formalized in the Coq proof assistant.

Static analysis approach can be also applied to catching covert channels. Doychev et al. [36] suggested a framework for the automatic, static analysis of cache side channels. In their approach, observed cache states, traces of hits and misses, and execution times are used to generate formal, quantitative security guarantees.

5. Checking Behavior

If the source code of software system is not available, or if annotating is not expressive enough to cover the safety feature of stakeholders, the safety of software must be determined by observing the behavior of software. Testing or monitoring is not complete, but it is useful to find some (but not all) defects of software system if testing or monitoring is used widely and intensively.

5.1 Fuzzing

Fuzzing or fuzz testing might be the most popular testing techniques in these days. It is one of the automated software testing techniques, and the software conducting fuzzing test is called fuzzer. Fuzzers provides unexpected or random data as inputs to a computer program, and monitors the program running for crashes or potential memory leaks. Depending on the existence of source code, fuzzing is classified as white-box fuzzing or black-box fuzzing [37,38]. Generating test cases randomly is less powerful than considering program running path in test case generation, but it is also known that fuzzing is much more cost-effective [39].

Godefroid et al. [40] proposed a grammar-based specification technique for generated random, but valid inputs for fuzzing. They also demonstrated that the complex structured-input guided by their proposed specification enhanced the effectiveness of white-box fuzzing. Stephens et al. [41] adopted symbolic execution when white-box fuzzing is performed. They showed that their fuzzer is efficient enough to identifying the same number of vulnerabilities, in the same time, as the top-scoring team of the qualifying event of the DARPA Cyber Grand Challenge.

5.2 Sandboxing

Sandboxing is a security mechanism for separating running programs, and protecting the programs from each other. It is usually used for executing untrusted or unverified code from outside of the system [42]. An untrusted program is put into a sandbox with a tightly controlled set of resources such as limited number of available APIs, restricted memory space, reduced network access or/and limited access to device drivers. In the sense of providing a highly controlled environment, sandboxing may be seen as a specific example of virtualization.

Sandboxing is considered as the most useful tool to controlling the behavior of JavaScript. Due to its popularity and weak semantics, running a program in JavaScript might make a host system very vulnerable. Van Acker and Sabelfeld [43] proposed JavaScript sandboxing systems considering JavaScript rewriting systems and browser modifications. Politz et al. [44] suggested a new type system for verifying sandboxing properties of JavaScript. They also implemented a light-weight verifier based on the proposed type system, and demonstrated the effectiveness of the proposed technique by applying it to *ADsafe*, which was reported to have several bugs and software weaknesses.

Agten et al. [45] proposed JavaScript sandboxing framework. Their framework requires no browser modifications: the sandboxing framework is implemented in JavaScript and is delivered to the browser by the websites that use it. Phung and Desmet [46] proposed a two-tier sandbox architecture to enable a website owner to enforce modular fine-grained security policies for untrusted third-party JavaScript code. The architecture consists of two layers: an outer sandbox that provides baseline isolation with generic, coarse-grained policies, and an inner sandbox that enables fine-grained policy enforcement specific to a particular untrusted application.

5.3 Information Flow Security

Preventing user information from owing to non-legitimate third-party gets more important than ever as more and more sensitive personal information is gathered into small smart phones. Covert channels, that means, leaking information through program termination, execution time, or exceptions, have become the major source of wrong information flow.

Since the source code of smartphone apps are not usually available for static analysis of potential information leaks, tracking or monitoring information flow is still the most viable ways of protecting the system from malicious information leaks. Enck et al. [47] implemented a system-wide dynamic taint tracking and analysis system, which is able to track multiple sources of sensitive data simultaneously. Their system also gave to users a visual analysis of how third-party applications collect and share their private data.

JavaScript have led the web with static text and images to a powerful application platform. Increasingly, web applications combine services from different providers. Despite of the handiness of JavaScript and its dialects, JavaScript is criticized by its security weakness as a programming language. Hendin et al. [48] tried to enhance the security level of JavaScript with fine-grained tracking of information flow. They implemented JavaScript interpreter, which tracks information in the presence of libraries, as provided by browser APIs, as well as enforces information-flow policies for the full JavaScript language.

5.4 Property-Based Random Testing

Property-based random testing (PBRT) is a form of black-box testing with formal statements of its intended behavior properties. The testing case is derived from the formal statements, and the software system is tested with a large number of random test cases. PBRT was popularized in the functional programming society by the *QuickCheck* library for Haskell [49]. Pacheco and Ernst [50] proposed PBRT for java called *RANDOOP*. Their system generates unit tests for Java code using feedback-directed random test generation. It also provides an annotation-based interface for specifying configuration parameters.

The concept of random testing and testing directed by feedback has improved to automated random testing. Godefroid et al. [51] proposed the concept of directed automated random testing. They argued that with their system, testing can be performed completely automatically on any program that compiles—there is no need to write any test driver or harness code.

6. Research Challenge

6.1 Trusted Computing Base

To guarantee the safety of implemented software, the underlying software systems must operate correctly. For example, if a compiler does not produce a correct code because of its internal bugs, all the care and attention a software developer pays would get useless. Some software such as compilers, operating systems or cryptography algorithms must be trusted, so they are called trusted computing base. There have been research cases showing the successful demonstration of verifying the trust-

worthiness of base software. For compilers, Leroy's CompCert project [6,52] demonstrates the viability of automatic or machine-assisted program verification. In this project, the correctness of an optimizing C compiler is specified, implemented and proved. They expressed the operational semantics of all the involved languages, source language and all intermediate languages, as inductive relations in the Coq [16,17] proof assistant. The Vellvm project [53] formalized the LLVM compiler's intermediate language and proved the correctness of some key optimizations.

In operating systems, the CertiKOS project [54,55] has built a new, fully verified and secure hypervisor kernel. Cerritos is hypervisor architecture that leverages formal certification to ensure correctness and counter information leakage in cloud computing. CertiKOS shows an example of applying recent advances in certified software design to implementation of a modular and evolvable certified kernel. Through machine-checkable proof certificates and runtime monitoring, CertiKOS provides users the assurance of correct and leak-free execution of their cloud services. Klein et al. [56,57] verified seL4 microkernel from an abstract specification down to its C implementation. A third-generation microkernel of L4 provenance, seL4, comprises 8,700 lines of C code and 600 lines of assembler, and they built a formal, machine-checkable verification. In their verification, they assumed correctness of compiler, assembly code, and hardware. It also shows that verification of individual software in software hierarchy must depend on trusted computing base.

6.2 Theorem Prover

Tools for reasoning about programs ranges from fully automatic program analysis tools, called *automated theorem prover*, to interactive tools where the human is closely involved in the proof process, called *proof assistant*. Early versions of theorem provers were first-order. The first-order theorem provers were applied to the problem of verifying the correctness of computer programs in languages such as Pascal, Ada, and Java *etc*. First-order theorem proving is one of the most mature areas of automated theorem proving. CARINE (Computer Aided Reasoning engINE) [58] and Larch Prover [59] are notable automated theorem provers in their performance and rigorous.

While other logics, such as higher-order logics, have more expressive power, theorem proving for these logics is far more difficult. Therefore, proving or/and checking higher-order logics are usually conducted by proof assistant systems with human involvement. Because of the flexibility and expressive power of higher-order logics, proof assistants are more prevalent than fully automated theorem provers: Coq [16,60], Isabelle [61,62], NuPRL [63], PVS [64], and Twelf [65,66].

6.3 Policy Specification

It is not very easy to determine which property will be essential to safe software not to intrude the system security. It depends on what is expected from a software system. It would be very useful for a user or diverse stakeholders to specify their policy requirement in a dedicated language, and then other parts such as certifying compilers and theorem provers can generate and check the proofs. REMS project [67] has calibrate formal specifications against observed behavior successfully. Formal languages like Z [68], Alloy [69], or AADL [70] are used for specifying properties of system models. Higher-order logic and type theories in modern proof systems will be successfully used for specifying arbitrary concepts as abstraction boundaries.

7. Conclusion

We have discussed the on-going approaches for guaranteeing or verifying the safety of software systems in this paper. We have also discussed that more vigorous research is needed in some areas such as assisting automatic or semi-automatic theorem proving, specifying arbitrary policy policies, and making the infrastructure software more trustworthy.

Building trustworthy software has been the aim of most of software developers since the dawn of computer science. Due to the intensive research on programming language semantics and formal verification as well as increasing hardware power and the ability of fast data analysis, we believe that we are close to machine-verifiable trustworthy software.

References

[1] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 1997, pp. 106-119.

[2] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Agents and Security*. Heidelberg: Springer, 1998, pp. 61-91.

[3] A. W. Appel, "Foundational proof-carrying code," in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, Boston, MA, 2001, pp. 247-256.

[4] A. W. Appel and D. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 5, pp. 657-683, 2001.

[5] J. Vanegue, "The weird machines in proof-carrying code," in *Proceedings of the IEEE Security and Privacy Workshops*, San Jose, CA, 2014, pp. 209-213.

[6] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107-115, 2009.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, pp. 527-568, 1999.

[8] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "Talx86: a realistic typed assembly language," in *Proceedings of ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta, GA, 1999, pp. 25-35.

[9] G. Morrisett, "Typed assembly language," in *Advanced Topics in Types and Programming Languages*. Cambridge, MA: MIT Press, 2005, pp. 141-176.

[10] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker, "Fault-tolerant typed assembly language," *ACM SIGPLAN Notices*, vol. 42, pp. 42-53, 2007.

[11] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, Palo Alto, CA, 2004, p. 75.

[12] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 2, article no. 6, 2015.

[13] H. Xi and R. Harper, "A dependently typed assembly language," *ACM SIGPLAN Notices*, vol. 36, no. 10, pp. 169-180, 2001.

[14] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Chalmers University of Technology, Goteborg, Sweden, 2007.

[15] L. Augustsson, "Cayenne: a language with dependent types," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, MD, 1998, pp. 239-250.

[16] B. Barras, S. Boutin, C. Cornes, J. Courant, J. C. Filliatre, E. Gimenez, et al., "The Coq proof assistant reference manual: Version 6.1," INRIA, *Report No. RT-0203*, 1997.

[17] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. Cambridge, MA: MIT Press, 2013.

[18] C. McBride, "Epigram: practical programming with dependent types," in *Advanced Functional Programming*. Heidelberg: Springer, 2004, pp. 130-170.

[19] E. Brady, "Idris, a general-purpose dependently typed programming language: design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, pp. 552-593, 2013.

[20] A. Jeffrey, "Dependently typed web client applications," in *Practical Aspects of Declarative Languages*. Heidelberg: Springer, 2013, pp. 228-243.

[21] G. Huet and H. Herbelin, "30 years of research and development around Coq," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 249-249, 2014.

[22] A. Athalye, "CoqIOA: a formalization of I/O automata in the Coq proof assistant," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 2017.

[23] S. Chatzikyriakidis and Z. Luo, "Natural language reasoning using proof assistant technology: rich typing and beyond," in *Proceedings of the EACL 2014 Workshop on Type Theory and Natural Language Semantics*, Gothenburg, Sweden, 2014, pp. 37-45.

[24] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002, pp. 55-74.

[25] D. Distefano, P. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in *Tools and Algorithms for the Construction and Analysis of Systems*. Heidelberg: Springer, 2006, pp. 287-302.

[26] J. Berdine, C. Calcagno, and P. O'Hearn, "Symbolic execution with separation logic," in Proceedings of *Asian Symposium on Programming Languages and Systems*. Heidelberg: Springer, 2005, pp. 52-68.

[27] J. Berdine, C. Calcagno, and P. O'Hearn, "Smallfoot: modular automatic assertion checking with separation logic," in *Formal Methods for Components and Objects*. Heidelberg: Springer, 2005, pp. 115-137.

[28] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 231-242, 2013.

[29] E. Pek, X. Qiu, and P. Madhusudan, "Natural proofs for data structure manipulation in C using separation logic," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 440-451, 2014.

[30] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: a practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*. Heidelberg: Springer, 2009, pp. 23-42.

[31] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts Essentials*. Hoboken, NJ: John Wiley & Sons, 2014.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 203-216, 1994.

[33] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *Proceedings of the IEEE 27th Computer Security Foundations Symposium*, Vienna, Austria, 2014, pp. 18-32.

[34] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5-19, 2003.

[35] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for C and assembly programs," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 648-664, 2016.

[36] G. Doychev, B. Kopf, L. Mauborgne, and J. Reineke, "Cacheaudit: a tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security*, vol. 18, no. 1, article no. 4, 2015.

[37] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ: Pearson Education, 2007.

[38] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA: Artech House, 2008.

[39] J. W. Duran and S. Ntafos, "A report on random testing," in *Proceedings of the 5th International Conference on Software Engineering*, San Diego, CA, 1981, pp. 179-183.

[40] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 206-215, 2008.

[41] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, San Diego, CA, 2016, pp. 1-16.

[42] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: confining the wily hacker," in *Proceedings of the 6th Conference on USENIX Security Symposium Focusing on Applications of Cryptography*, San Jose, CA, 1996.

[43] S. Van Acker and A. Sabelfeld, "JavaScript sandboxing: isolating and restricting client-side JavaScript," in *Foundations of Security Analysis and Design VIII*. Cham: Springer, 2015, pp. 32-86.

[44] J. G. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: type-based verification of JavaScript sandboxing," in *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, 2011,

[45] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, Orlando, FL, 2012, pp. 1-10.

[46] P. H. Phung and L. Desmet, "A two-tier sandbox architecture for untrusted JavaScript," in *Proceedings of the Workshop on JavaScript Tools*, Beijing, China, 2012, pp. 1-10.

[47] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for real time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, article no. 5, 2014.

[48] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: tracking information ow in JavaScript and its APIs," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Gyeongju, Korea, 2014, pp. 1663-1671.

[49] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *ACM SIGPLAN notices*, vol. 46, no. 4, pp. 53-64, 2011.

[50] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, Montreal, Canada, 2007, pp. 815-816.

[51] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213-223, 2005.

[52] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 42-54, 2006.

[53] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 427-440, 2012.

[54] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo, "CertiKOS: a certified kernel for secure cloud computing," in *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, 2011.

[55] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 595-608, 2015.

[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, MT, 2009, pp. 207-220.

[57] G. Klein, J. Andronick, G. Keller, D. Matichuk, T. Murray, and L. O'Connor, "Provably trustworthy systems," Philosophical Transactions of the Royal Society A, vol. 375, no. 2104, 2017.
[58] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97-109, 1985.
[59] S. J. Garland and J. V. Guttag, "An overview of LP, the Larch Prover," in *Rewriting Techniques and Applications*. Heidelberg: Springer, 1989, pp. 137-151.
[60] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, "CertiCoq: a verified compiler for Coq," in *Proceedings of the 3rd International Workshop on Coq for Programming Languages*, Paris, France, 2017.
[61] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Heidelberg: Springer, 2002.
[62] M. Wenzel, "Isabelle as document-oriented proof assistant," in *Intelligent Computer Mathematics*. Heidelberg: Springer, 2011, pp. 244-259.
[63] P. B. Jackson, *The Nuprl Proof Development System (Version 4.2) Reference Manual and User's Guide*. Ithaca, NY: Cornell University, 1994.
[64] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas, "PVS: combining specification, proof checking, and model checking," in *Computer Aided Verification*. Heidelberg: Springer, pp. 411-414, 1996.
[65] F. Pfenning and C. Schurmann, "System description: Twelf: a meta-logical framework for deductive systems," in *Automated Deduction (CADE-16)*. Heidelberg: Springer, 1999, pp. 202-206.
[66] C. Schurmann, "The Twelf proof assistant," in *Theorem Proving in Higher Order Logics*. Heidelberg: Springer, 2009, pp. 79-83.
[67] P. Sewell, "REMS: rigorous engineering of mainstream systems," [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/.
[68] J. Madey, "Book Review: the Z notation: a reference manual: JM Spivey. Prentice Hall International, Hemel Hempstead, United Kingdom, 1989," *Science of Computer Programming*, vol. 15, no. 2/3, pp. 253-255, 1990.
[69] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, 2012.
[70] P. H. Feiler and D. P. Gluch, *Model-based Engineering with AADL: an Introduction to the SAE Architecture Analysis & Design Language*. Upper Saddle River, NJ: Addison-Wesley, 2012.

Hyong-Soon Kim https://orcid.org/0000-0002-6144-3791

He received B.S., M.S., and Ph.D. degrees in Department of Computer Science and Engineering from Korea University in 1995, 1997, and 2009, respectively. Since 1997, he has been with National Information Society Agency of Korea as an Executive Principal Researcher.

Eunyoung Lee https://orcid.org/0000-0001-8703-9730

She received her Ph.D. degree in Department of Computer Science from Princeton University in 2004. Since 2005, she has been with Dongduk Women's University, Korea as faculty. Her current research interests include software security, parallel computing, and cloud computing.