Towards Trusted Systems from the Ground Up

Vivek Haldar
Information and Computer Science,
University of California, Irvine
CA 92697-3425, USA
vhaldar@uci.edu

Michael Franz
Information and Computer Science,
University of California, Irvine
CA 92697-3425, USA
franz@uci.edu

ABSTRACT

Operating systems, the most fundamental software layer in virtually every computer system, are notoriously insecure and unreliable. A possible reason for this situation is that progress on language-based safety and security mechanisms has largely been ignored in the context of operating systems. There is a lack of *mechanical checking of safety properties* (both at compile- and run-time) as well as a framework and a mechanism for expressing, safely transporting and enforcing such properties. Our solution is to leverage language-based mechanisms by reversing the traditional relationship of operating systems and programming languages – implement operating system functionality on top of a provably safe and secure language and its runtime environment instead of the other way round. We propose to leverage these mechanisms, many of which have been developed in the context of mobile code infrastructures, to build secure systems from the ground up. Such a system would be more secure, flexible and scalable compared to existing systems.

1. INTRODUCTION

The traditional view of computer systems – that is still prevalent today – is that of an operating system abstracting away hardware details and providing basic services such as process management, memory management etc. All applications are built on top of the operating system, utilizing the services it provides. Perhaps the most critical of these applications is the compiler, which translates some high level language to native executable code.

The failing of this model is that even modern operating systems are written in unsafe languages such as C. Buffer overruns and stack smashing are still by far the most common causes of security breaches in modern systems[18]. Trust in an operating system is established with time, as more and more people are able to use it without major problems and more bugs are discovered and weeded out. However, there is no concept of mechanically being able to check the operating system for compliance with a security policy. Thus it is not possible to make any security claims about modern operating systems other than "it's been out there for two decades and it seems to work". This problem is starkly highlighted when automated tools such as Engler's meta-compilation[3] can find hundreds of bugs in operating systems that have been in wide use for a long time and were believed to be relatively bug-free.

At the same time, language-based techniques can now express and check a number of high-level safety properties. Examples of such properties are: "acquired locks must be released[3]", information flow[23] and "some operations must be done in a certain order[21]". Also, research in mobile code has developed techniques to *safely transport* these properties as part of some intermediate representation.

In spite of many promising results in language level approaches to security, traditional systems only offer those security advantages from the "OS upwards", because the OS is still in a fundamentally unsafe, weakly typed language (typically C).

The rest of the paper is structured as follows: after giving a very broad introduction to language-based security in section 2, we present our proposed system in section 3. Section 4 presents related work, and section 5 concludes.

2. LANGUAGE-BASED SECURITY AND MOBILE CODE

Language-based techniques for expressing and checking security properties have been gaining importance as a viable and effective way to write secure programs. The case for language-based security has been made elsewhere[15][19] – here we simply give a broad overview of the major techniques and summarize its major advantages.

A *safe* programming language provides certain guarantees about the execution of programs written in that language[16][17]. These include at least the following: *memory safety* – the program reads and writes memory correctly, respecting types, bounds and access restrictions; *control flow safety* – the program does not jump to arbitrary locations (e.g. a location that does not contain code); and *type safety* – operations and function calls receive arguments of the correct type.

In modern programming languages, safety is guaranteed by a combination of static and dynamic checks. *Type systems*[16] are an extremely effective method to rule out a large number of run-time errors using a method that is well understood, has a solid theoretical foundation and where checking is efficient in practice.

Lack of safety in programming languages is a major source of security violations in computer systems. A survey of various well-known bug tracking resources shows that almost *half* of all exploited vulnerabilities in software can be blamed on buffer overruns[18].

The *language-based*[15][19] approach to security has been variously defined as "a set of techniques based on programming language theory and implementation…brought to bear on the security question", and "leveraging program analysis and program rewriting to enforce security policies". These techniques can be thought of as falling into two major categories – program analysis and program rewriting.

Program analysis covers a variety of techniques that statically try to check a program's conformance to a security policy. Examples are types-based approaches to security such as typed assembly language (TAL)[20].

Program rewriting is a complementary set of techniques that enforce security policies by rewriting programs to conform to them. Inlining security monitors[21] is an example of this. Another example is proof-carrying code (PCC)[14] in which the compiler, along with object code for a program, emits a *certificate* of proof for conformance with a security policy. Though such proofs might be hard to generate, they are very easy to verify.

Each approach has its advantages and disadvantages, and a comprehensive, flexible, expressive and powerful security architecture would need to combine all three elements in a thoughtful manner.

The primary advantage of language-based security is that it is flexible and can easily express very fine-grained policies. For example, the memory protection policy of a type-safe program is very fine-grained compared to the coarse page-level protection that modern operating systems typically provide. Also, since the programmer deals with language-level semantics while writing a program, it is much more useful and intuitive to raise security concerns to the language-level rather than leave them in other parts of the system.

To transport secure code between machines, we need to encode programs in a portable, safe, and tamper-proof manner; these goals are identical to those of mobile code infrastructures. Mobile code has been an extremely active area of research spurned by the ubiquity of the Internet and the consequent need for running third-party untrusted code. One outcome of this research has been a range of techniques for ensuring the *security* of mobile code, especially by using specialized *mobile code formats*. These mobile code formats are amenable to security checks. Thereafter, they could be interpreted or compiled to native code for execution. Examples are the popular bytecode-based Java format[24], as well as other higher level representations such as abstract syntax trees[25]

Note that even though secure mobile code formats were developed in the context of running untrusted code over the Internet, their security advantages are just as attractive for running *any* code. In fact, we propose that in order to have a truly trusted system, we should do away with native code altogether, and *all* code in the system should be in some secure format. Native code would only be generated when needed.

3. A WHOLE SYSTEM BUILT ON SECURE CODE

Our goal is to build a practical system about which we can make security guarantees from the *ground up*, and not just from the operating system up. We assume that all hardware (or at least the CPU) is part of the trusted computing base, and start from there. We propose making use of language-based security to build a trusted system from the level of hardware and above.

We propose to do this by essentially inverting the roles of compiler and operating system in the traditional view of computer systems. Instead of the compiler and other applications being conceptually built on top of the operating system, we now have the compiler as the *first layer* above hardware. The function of this compiler is to translate some secure, type safe intermediate representation (IR) to native code. The *whole system*, including the operating system, is now built on top of this compiler. It remains to be investigated what intermediate representations are suitable for this, and whether it would be best to have one intermediate representation or support a number of them. It also remains to be investigated what the right mix of static and dynamic checks is.

The proposed system would utilize a compiler to compile high-level programs down into a secure type-safe intermediate representation. A second compiler, which essentially sits in the place that traditionally would be occupied by the operating system, would translate this intermediate representation code to native machine code at the time of execution. Note that that for most secure intermediate representations only the second compiler (from IR to native code) needs to be part of the trusted computing base.

The advantage of this approach is that it enables us to leverage all the existing research on language based security and apply it to the design of a secure system from the hardware up. When the whole system is built on top of a secure type-safe IR, it is possible to reduce most security concerns to the language level, where the compiler can automatically check them. A pleasant by-product of this is *portability* since all code is in some machine-independent intermediate representation.

One challenge here is to build this system with the absolute minimum of unchecked code that has to be trusted, while retaining acceptable performance.

3.1 A Typed View of Hardware

We also propose to carry types down to the level of hardware by exposing the hardware to the upper layers of the system in a *strongly typed* manner. We call this a typed hardware abstraction layer or *typed-HAL*. This is in keeping with our philosophy of using language-level security mechanisms to the fullest. By exposing hardware only as a typed interface, and *disallowing arbitrary operations on it*, it becomes all the more difficult to subvert the system. Moreover, it does away with a large number of potential bugs in systems that deal with hardware in a raw, untyped manner.

For example, consider the common activity of handing down a network packet for transmission to the network card in a PC. The network card essentially views this packet as a *raw sequence of bytes*. However, note that this packet is *not* just a raw sequence of bytes – it is required to have a very particular structure, as dictated by the particular network protocol being used, say TCP/IP. In modern operating systems, *no check is done to make sure the packet has this particular structure*. The fact that the packet is well formed is taken for granted because it was formed by another part of the operating system, which is trusted[1]. Packets that are intentionally constructed to be malicious may exploit loopholes. One conspicuous example of this was the Ping Internet exploit (also called ``The Ping of Death"[13]) that could crash a remote machine simply by sending a specially constructed packet to it. There are no automatic mechanical checks. Essentially, we trust the programmer who implemented the network protocol stack of the operating system.

Such problems would vanish in a system that took a typed view of hardware and the data that is sent in and out of it. In the example given above, had the operating system and network protocol stack been written in a strongly typed language, it would be possible to define the structure of the packet as a type. Combined with a typed-HAL, that would only expose the network card as a *typed interface*, we would be able to easily do away with bugs in the structure of network packets by virtue of strong type checking *that is automatically done by the compiler*.

For another example, consider the problem of buffer overruns, which is by far the most common cause of security breaches. Strongly typed language runtimes would prevent such exploits. Strong typing would not prevent a programmer from indexing an array outside bounds, but it would catch the illegal access at runtime.

If we ask the question – *what is the basis for insecurity in a system?* – The answer is deceptively simple. Fundamentally,

[1] The problem is even more severe when third party modules are inserted into the kernel to extend it.

insecurity arises from *doing operations that are not supposed to be done*. Then ask the question – *what are types used for?* – And the answer is: *making sure that only allowed operations are performed on data items of the correct type*. From these two observations it is easy to see the motivation for proposing a typed hardware abstraction layer. By exposing the hardware *only* as a typed interface, we seek to disallow arbitrary operations and limit the operations possible by using hardware to only the legal ones. This does away with a major source of security breaches in one clean swoop. In essence, the programmer sees only a typed view of the machine. This model of the machine is already familiar to programmers using typed high-level languages such as Java.

4. RELATED WORK

Our approach is similar in spirit to a number of operating system projects that built entire operating systems using type safe languages.

SPIN[1] is an operating system written in Modula-3. It uses Modula's type safety and encapsulation properties to enforce safety, modularity and protection between applications. Extensions to the OS are also written in Modula. This makes it possible to add extensions to the operating system in a safe manner, because they are type checked at compile-time, which implies their safety. Type safety and encapsulation are also used to enforce separation between logical domains by using separate namespaces, a language-level feature. However, the main objective of the designers of SPIN was to create an operating system that could be safely extended to meet the specialized requirements of applications. As such, the use of Modula's safety features is relied upon only for extensions within the kernel, and not for the system as a whole.

The Oberon system[2][10] is an entire system written in the language of the same name. Its main emphasis was on extensibility of the system. Safety was guaranteed by the fact that everything in the system was written in Oberon, a strongly typed language. The Oberon system turned out to be surprisingly portable and was implemented on a large variety of hardware platforms.

In current mainstream operating systems such extensibility is usually through some sort of module or device driver mechanism that can add new functionality to the kernel or support a new hardware device. But these modules are not checked for safety and are essentially trusted by the operating system. Since these modules run inside the kernel in privileged mode they can access any data in the system. A malicious module has free reign in the system. A bug in the module affects the entire system. If a module crashes it takes the entire system with it.

In the Oberon and SPIN systems, safety is attained by virtue of using a type-safe language. This is what makes it safe to insert extensions into the operating system. However, this has the disadvantage of adding the compiler for that language to the trusted computing base, because we must trust the compiler to emit code that conforms to the type system of the language and has been checked properly.

A number of systems, such as JavaOS[26] and JX[27], have implemented a Java virtual machine on bare hardware, with some OS-like functionality such as processes.

Our approach is much broader, since we propose using the full gamut of static as well as dynamic checks to express and enforce high-level safety and security properties, and not just those expressible in the Java bytecode model. We would also like to explore the design space for mobile code representations beyond bytecode.

In another approach, safety of extensions is ensured not by type-safety of a language but by a combination of software fault isolation[9] and transaction monitoring. An example of this is the Vino system[6][7], which uses software fault isolation to enforce safety of memory accesses. In order to prevent an extension (which in the Vino system is called a *graft*) from unduly holding resources their execution is treated as a sequence of *transactions* that the kernel keeps track of. Whenever an extension oversteps its bounds, (be it holding the CPU for too long, or allocating too much memory) the kernel terminates its execution and simply unwinds to the state at the time of the last transaction. This technique has a considerable overhead and slowdowns of up to 200% have been reported[7].

Note that the checks that are done by software fault isolation become unnecessary in a strongly typed language. As the system designers of Vino themselves concede, using a typed language would have saved them much implementation effort and would probably have resulted in a more efficient system.

The Exokernel project[4] is based on the assumption that operating system abstractions get in the way of efficient implementation of applications. This is because operating system abstractions are designed to be general and more often than not are not a good match with the specialized requirements of a particular application. This can lead to poor performance. The aim of an exokernel is to provide a very thin, minimal abstraction of hardware, as devoid of policy as possible. The operating system is then simply reduced to a user-level library. The major performance gain comes from eliminating most of the overhead of context switches between user and kernel mode, and improvements of up to four times have been reported for typical applications.

Our proposal would share the advantages of the exokernel approach, since we also propose exporting a very thin, minimal interface of hardware to the rest of the system. But in addition, one of our primary goals is security, and unlike the exokernel project, we propose exporting a typed view of hardware.

The Flux research group at the University of Utah has built a modular, component-based toolkit for building operating system, called OSKIT [5]. Many language researchers have used OSKIT to port implementations of various high level languages to run directly on hardware, as opposed to running them on top of an existing operating system. Their primary goal, however, is to explore how high level language level mechanisms (such as continuations [8]) can be efficiently implemented on hardware, without a policy-laden operating system getting in the way. Also, since OSKIT itself is implemented in C, it suffers from the same problems that we pointed out.

An example of using types for enforcing security properties is *packet types*, proposed independently by Sekar et al [12] and Chandra et al[11]. They use the concept of types to check the structure of network packets. They define a small domain-specific language for defining the structure of network packets. This language is strongly typed. It uses inheritance to capture the idea of protocol stacks with nested packet structures. Packet definitions in this language are used to automatically generate code that can parse and check incoming packets for conformance. An added advantage is the ability to concisely do pattern matching on packets. Sekar et al use this to detect low-level network attacks[12]. Though they use a domain-specific language, most of the notions used can be carried over easily to modern strongly typed languages such as Java. However, some special runtime support would be needed

5. SUMMARY

We note that most security breaches in modern computer systems are a consequence of weak typing. This is because almost all modern operating systems are implemented in a weakly typed language such as C in which most checks are left to the programmer, and are not mechanically enforced. The field of language-based security offers many promising solutions to the problem of specifying security policies, checking conformance against them and enforcing them.

We propose to build a trusted system on tamper-proof trusted hardware, from the ground up, by leveraging a combination of language-based techniques for security. We envision all code in the system, including the operating system, being in some type-safe, secure intermediate representation. At the heart of our system is the idea of bringing language-based security into the operating system kernel itself, eliminating a substantial source of security breaches. We propose to do this by implementing on bare hardware a compiler for a secure type-safe intermediate representation. The rest of the system would use the abstractions provided by this compiler kernel. Also, we propose to export only a strictly typed view of hardware to the rest of the system. We call this a typed hardware abstraction layer. This again eliminates another important source of security breaches, by constraining the ways in which hardware can be used.

We see the following as novel contributions of the proposed research:

Security from the ground-up: starting from hardware that is trusted, building a secure system from the ground up, and not just from the operating system-up, as is the case with current systems.

Typed hardware abstraction layer: exporting only a typed view of hardware to the rest of the system, which would automatically disallow illegal operations on hardware and ensure data integrity.

Building a whole system from mobile code: leveraging language based security right from the level of hardware, and implementing the entire system, including the operating system in a secure type safe intermediate representation, enabling us to make safety guarantees about it.

ACKNOWLEDGMENTS: Thanks are due to Niall Dalton, Peter Froehlich, Peter Housel, Fermin Reig, Christian H. Stork and Alex Strashny for many fruitful discussions and comments.

6. REFERENCES

[1] B. Bershad, S. Savage, P. Pardyak, E. Gunder, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety, and performance in the SPIN operating system. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pp. 267--284, 1995.

[2] M. M. Brandis, R. Crelier, M. Franz, and J. Templ. The Oberon System Family. Software—Practice and Experience, 25(12):1331–1366, Dec. 1995.

[3] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, CA, 23–25 Oct. 2000.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In Symposium on Operating Systems Principles, pages 251–266, 1995.

[5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In Symposium on Operating Systems Principles, pages 38–51, 1997.

[6] Y. E. James. Vino: The 1994 fall harvest.

[7] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In Operating Systems Design and Implementation, pages 213–227, 1996.

[8] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In Second ACM SIGPLAN Workshop on Continuations, Jan. 1997.

[9] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203–216, 1993.

[10] N. Wirth and J. Gutknecht. Project Oberon. Addison-Wesley, 1992.

[11] S. Chandra and P. J. McCann. Packet Types. In Second Workshop on Compiler Support for Systems Software (WCSSS), May 1999

[12] R. Sekar, Y. Guang, S. Verma and T. Shanbhag A High-Performance Network Intrusion Detection System. In ACM Symposium on Computer and Communication Security, 1999.

[13] CERT Advisory CA-1996-26. Denial-of-service attack via ping. http://www.cert.org/advisories/CA-1996-26.html. October 1996.

[14] G. C. Necula. Proof-carrying code. In Conference Record of POPL'97: The 24th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pages 106–119, Paris, France, January 15–17, 1997.

[15] D. Kozen. Language-based security. In Mathematical Foundations of Computer Science, pages 284–298, 1999.

[16] B. C. Pierce. Types and Programming Languages. MIT Press, 2002. To be published.

[17] L. Cardelli. Type systems. In Allen B. Tucker, editor, Handbook of Computer Science and Engineering. CRC Press, 1996.

[18] D. Wagner, J. S. Foster, E. A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Network and Distributed System Security Symposium, San Diego, CA, pages 3–17, February 2000.

[19] F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In Informatics, pages 86–101, 2001.

[20] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. ACM Transactions on Programming Languages and Systems, 21(3):527–568, May 1999.

[21] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In New Security Paradigms Workshop, pages 87–95, Ontario, Canada, 22–24 September 1999.

[22] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, pages 59–69, Snowbird, Utah, June 20–22, 2001. SIGPLAN Notices, 36(5), May 2001.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages [POP99], pages 228–241.

[24] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[25] C. H. Stork and V. Haldar. Compressed Abstract Syntax Trees for Mobile Code. In Workshop on Intermediate Representation Engineering (IRE 2001), July 2001, Orlando, Florida

[26] T. Saulpaugh and C. Mirho (1999) Inside the JavaOS Operating System. Addison Wesley, Reading, Massachusetts.

[27] M. Golm, J. Kleinöder, and F. Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In HotOS 2001, May 20-23, 2001, Elmau/Oberbayern, Germany