

Safe Code—It’s Not Just For Applets Anymore

Michael Franz

University of California, Irvine

Abstract. Despite numerous “trusted systems” initiatives, current systems software continues to be riddled with errors. The recent “Slammer Worm” incident shows that an adversary exploiting just one such error (a “buffer overrun”) can cause tens of thousands of hosts to fail around the world in just minutes. Even more suprisingly, for this particular vulnerability there had been a patch available more than six months earlier, and yet even many of the OS manufacturer’s (Microsoft’s) own computers succumbed to the attack.

Two facts are becoming increasingly evident: First, operating systems (and increasingly also application programs) are becoming so large and evolving so quickly that it is getting prohibitively expensive to manually inspect every line of code for the absence of errors. Second, the current approach of “patching” errors as they are discovered is failing us—on one hand, we now have attacks that can spread worldwide in minutes. On the other hand, if not even Microsoft can keep its own computers current, then how can one expect that other organizations will apply all patches immediately as they are released, and in the correct order?

While the situation looks dire in the short term, there is encouraging news for the longer term: Recent research results from the mobile-code community indicate that the underlying problem can be solved in a fundamental manner, using type-safe programming languages in conjunction with code verification techniques. In many mobile-code systems, incoming programs from untrusted hosts are *verified* prior to execution. Verification means that the code *itself* is automatically examined before it is executed—this is very different from (cryptographic) *code signing*, in which only the transport envelope of the code is checked and the code is trusted blindly if the check succeeds.

We envision a future in which even the operating system is verified *prior to every execution* and have begun implementing such a solution as a proof of concept. The only thing that needs to be trusted in our architecture is a minimal safe-code platform *core* (encompassing a verifier and a small dynamic code generator), small enough to be manually audited using techniques appropriate for mission-critical software, such as fly-by-wire control systems. The core is sealed along with the processing unit into a tamper-proof hardware implement. Everything above this layer is verified, i.e., even the code in the root directory of the local hard drive need no longer be trusted.

1 Background

In January 2003, the fastest-spreading self-propagating Internet attack to date infected nearly 90% of all vulnerable hosts worldwide in under *ten minutes*. The tiny program at the heart of the attack (later called the “Slammer worm”) consisted of just 376 bytes of code, yet it caused crippling slowdowns, brought down vital services (airline reservation systems, automatic teller machines), and rendered the Internet virtually unusable.

What finally slowed its spread were not active counter-measures taken against it, but the network outages it created in the first place.

In the aftermath, a thorough analysis [MPS⁺03] revealed the following astonishing facts:

- The number of infected hosts doubled almost every *eight seconds*.
- The attack exploited a buffer overflow vulnerability in Microsoft's SQL Server database product. This particular vulnerability had been made public six months prior to the attack, and a patch had been released even earlier.
- A number of Microsoft's own servers were affected by the attack.

Devastating as it already was, the attack could have been far worse. This particular worm carried no malicious "payload", i.e., its only purpose was propagating itself. One can only imagine what the damage would have been had it also deleted files, leaked database contents, or any of the other capabilities that had been well within its reach.

What lessons can be learned from this incident? First, the current practice of "patching" vulnerabilities as they are discovered clearly does not work, as the example of Microsoft's own infected servers shows. If the world's largest software manufacturer cannot even keep the patching regime for its own products on its own servers, then how can one expect compliance in any other large organizations? Second, the blinding speed with which the worm spread rendered real-time human intervention ineffective. By the time anyone could even think about a "patch", the worm had long already run its course.

"Slammer" was not a first of kind. In mid-2001, the "Code Red" (or "Nimda") worm utilized very similar techniques. It also exploited a buffer-overflow vulnerability, this time in Microsoft's IIS web server. However, it had a much slower infection rate, doubling the number of infected hosts only about every 37 minutes. The prototype for this kind of attack was the worm that Robert Morris, Jr. released on the Internet on November 2, 1988, exploiting a buffer-overflow in the Unix `fingerd` daemon. It affected between 5%-10% of all hosts on the Internet at that time.

The common denominator in all these attacks (and numerous others) lies in the exploitation of an unsafe programming language. In all three above cases, the language was C, and the exploited vulnerability was its inability to range-check array accesses. The CERT/CC Advisories for 2001 [cer01] enumerates a number of vulnerabilities in essential communications-, data storage-, and system administration software. This list includes software from Microsoft, Oracle, IBM, Hewlett-Packard, Cisco, as well as open source code. Most of the items in this list are buffer overflow and format string vulnerabilities. These vulnerabilities only exist because of the lack of safety in the C programming language. A survey of various well-known error tracking resources publicly available on the Internet shows that almost *half* of all exploited vulnerabilities in computers stem from buffer overruns [WFBA00].

The actual situation is even more ominous than it initially appears since buffer overflows are only the most primitive and obvious attacks on systems written in a weakly typed unsafe language such as C. They are also perhaps the easiest to detect using static analysis. There could easily be a multitude of more sophisticated attacks that exploit weak typing loopholes *other* than buffer overruns. As things stand, it is impossible to determine how many of these vulnerabilities are bona fide programming errors and how many are intentional "back doors" placed by agents of foreign nation-states working for

domestic software companies. It should be assumed that several foreign intelligence services perhaps have partial “maps” of exploitable vulnerabilities in software systems. For such a foreign service, nothing worse can happen than a “hacker” accidentally stumbling across such a carefully placed vulnerability, destroying its strategic value.

2 Toward A Solution: Insights From The “Mobile Code” Domain

A number of techniques have been proposed to counter the lack of safety in common infrastructure software written in C [CPM⁺98]. Techniques that insert run-time checks [ABS94], static analyses [WFBA00] and combinations of both [NMW02] have been applied to the problem of checking pointer safety in existing C programs. These efforts have uncovered countless errors in common C programs such as the Linux Net Tools and SPECInt benchmarks—including errors that had earlier been overlooked in specific hand audits by humans searching for exactly these safety holes.

While the above techniques are fairly effective in the short-term, the underlying philosophy in most of the software systems communities (operating systems, control software, etc.) is still one of “adding patches when an error is discovered”, rather than trying to solve the basic underlying problem—the lack of type-safety.

Current commercial efforts at providing a more trustworthy computing base, such as Intel’s *LaGrande* and Microsoft’s *Next Generation Secure Computing Base* (also called *Palladium*), will not fundamentally improve the situation. These solutions are based on hardware-assisted *code signing* using cryptographic methods. While they prevent the execution of malicious code from unauthorized providers (and can be used to extort licensing fees from application developers wishing to become “authorized”), these trusted computing platforms are just as vulnerable to exploitation of programming errors as current operating systems are. And hence, we will inevitably see the equivalent of the “Slammer Worm” happening even under these systems.

Conversely, there exists a domain in which the problem has been addressed more fundamentally, namely the “mobile code” context. A considerable amount of effort has recently been invested into mobile code research. An important focus of this research has been the *safety of code supplied by an untrusted party*. Unlike cryptographic envelope techniques such as code signing, many mobile-code systems attempt to *verify* all mobile code prior to execution. Verification means determining the code’s safety *by examining the code itself* rather than where it came from. In the following, we apply these ideas much more broadly than just for “mobile code”.

Over the past few years, three main approaches have emerged for establishing the safety of code supplied by an untrusted (and potentially malicious) third party. They are, in turn, *virtual machines with code verification* [LY99], *proof-carrying code* [Nec97], and *inherently safe code formats* [ADvRF01,HSF02,ADF⁺01].

- In virtual machines with code verification, the code is examined to ensure that the semantic gap between the source language and the virtual machine instruction format is not exploited. For example, virtual machines have general *GOTO* instructions but not all possible control flows are actually legal. As another example, the language definition of Java requires every variable to be initialized before its first use.

Unless control flow is strictly linear, this property cannot be inferred trivially from the virtual machine program but requires the verifier to perform dataflow analysis.

- In proof-carrying code solutions, the code producer attaches a *safety proof* to an executable. Upon receiving the code, the recipient examines it and calculates a *verification condition* from the code. The verification condition relates to all the potentially unsafe constructs that actually occur in the executable. It is the task of the code producer to supply a proof that discharges this verification condition, or else the code will not be executed.
- In the third approach, an *inherently safe code format* is used to transport the mobile program, making most aspects of program safety a well-formedness criterion of the mobile code itself. Checking the well-formedness of such a format is much simpler than verifying bytecode. However, it requires a much more memory-intensive machinery at the code recipient’s site, since inherently safe formats are based on compression using syntax and static program semantics. As a consequence, this approach is less well suited for resource-constrained client environments.

Clearly, these approaches are one step ahead of the current safety practices in general software systems. They are becoming increasingly important since *all* code is fast becoming “mobile”, with program patches and whole applications increasingly distributed via the Internet. However, mobile-code techniques as they currently exist are not [yet] suited to support large applications. More fundamentally, all of the existing mobile-code solutions tacitly assume that they execute on top of a trusted and correct operating system (Figure 1a). Our research is aimed at removing these two drawbacks.

3 FSSA: A Foundational Safe System Architecture

Any non-trivial system today consists of several layers, starting with the hardware on the very bottom, and typically an operating system right on top of that. The trouble with layered systems is that, similar to buildings, one cannot build something stable on top of a rotten foundation. In particular, one cannot build a secure system on top of an insecure or faulty operating system.

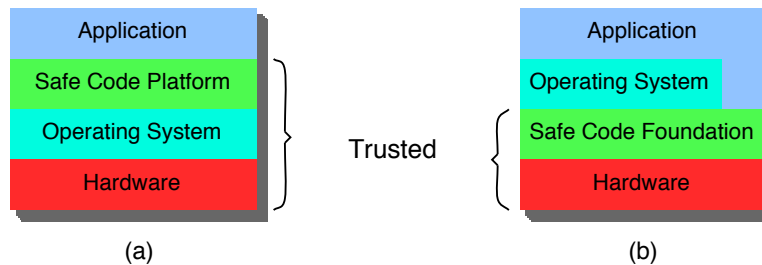


Fig. 1. (a): Typical architecture of a modern system supporting “mobile code”. Note that the operating system is trusted, along with the safe code platform itself. (b): Foundational Safe System Architecture. Here, the operating system is no longer part of the trusted computing base.

The problem with today's operating systems of course is that on the one hand they are very large, making it difficult to perform meaningful human audits, and on the other hand they evolve rapidly or even dynamically (in terms of downloading and opening dynamic link libraries while the system keeps running), making it difficult to use static code inspection tools.

In practice, trust in a particular implementation of an operating system is established with time, as more and more people are able to use it without major problems and more and more errors are discovered and weeded out. There is no concept of mechanically being able to check the system for compliance with a security policy. Hence, it is not possible to make any security claims about large operating systems other than "it's been out there for two decades and it seems to work". This problem becomes apparent when automated tools such as Engler's meta-compilation engine [ECCH00] are able to find hundreds of errors in operating systems that have been in wide use for a long time and were believed to be relatively correct.

But at the same time, we are now adding functionality to existing operating systems at such a pace that no part of any current mainstream operating system is really old enough to have reached the level of maturity at which one can assume that it is really "mostly correct". Add to this the new threat of system-level programmers in the employ of foreign nation-states that might be planting new backdoors in code that was thought to be mostly stable.

More recently, there have been promising attempts to shore up existing operating systems, making them less susceptible to errors. The most notable of these efforts are "Trusted BSD" [BSD], "Trusted Debian" [Deb], and NSA's Security-Enhanced Linux [NSA]. These have been able to remove some, but not all the shortcomings from their respective operating system ancestors. However, while useful and notable, these efforts will not be able to mend the more fundamental dual problems of OS size and evolution speed.

We believe the long-term solution to all these problems can only lie in *no longer trusting the operating system*. Given our deep involvement in mobile-code research, we are confident in claiming that a system can be built in which all of the operating system can be verified mechanically before execution, much in the same way that a mobile program is verified in today's mobile code systems. In such a system (Figure 1b.) there would only be a minimal and stable piece of code (the "Safe Code Foundation") on top of the processor that would need to be trusted, small enough to be audited by humans to the highest standards. We call this approach "foundational" because safety is engineered into the system at the foundation.

4 How To Avoid Having To Trust The Operating System

We are currently working on a comprehensive security architecture that uses language-based mechanisms to eliminate errors due to circumvention of type safety, be they intentional or erroneous, and that additionally uses security policy mechanisms to contain malicious behavior. Our approach extends techniques previously applied to mobile code and is based on a combination of a) mechanically verifying the absence of such errors

in any software before it is run, and b) monitoring executing software for malicious activity.

Note that language-based type safety is not the same as complete absence of run-time errors. However, run-time errors in a type safe environment do not cause spurious behaviour and cannot compromise the integrity of a system. Detection of a type error may cause abrupt program termination, but cannot be exploited for any other purpose. Also, the cause of the error is traceable in terms of constructs of the programming language—for instance “division by zero”—as opposed to details of the implementation, such as “bus error”.

The main innovation of mobile-code systems, popularized by Java, is a concept of code security based on *examining the code itself*. Previous concepts of security had been based on access control (physical access, accounts, file ownership, etc.) and on cryptographic authentication (code signatures, etc.). The concept of examining the code itself to determine if it is safe gives rise to a new class of defenses against adversaries, complementing the other two (access control and authentication). The general public has largely misunderstood this, thinking that mobile code needs verification because it has an inherent defect. Nothing could be further from the truth—mobile code is showing the way to making code safer *in general*, by providing this third pillar of security in addition to the other two. For example, one could sign Java programs with a cryptographic signature, if one so desired.

Our FSSA approach takes this idea of verification from Java and similar mobile-code platforms, and puts the whole operating system on top of such a type-based safe-code platform. However, this is where the similarity ends. One cannot merely use, for example, the existing Java Virtual Machine (JVM) and simply implement the operating system on top of it. First, this would make the operating system far too inefficient. Second, the virtual machine would also be fairly large, leading back to the same problems of scale that plague today’s operating systems.

Instead, our solution takes an equal part of inspiration from recent advances in Proof Carrying Code (PCC) research. Alas, existing proof-carrying code techniques have a different significant shortcoming, namely that the generated code is not portable. Once a program has been translated from source code into its executable format, the target architecture cannot be changed. Hence, using existing PCC techniques, one cannot create a long-lived and efficient architecture that can survive many generations of hardware evolution. Also, the proofs that result from this technique can be very large, much larger than the programs they relate to. One of the reasons why PCC has these very large proofs is because the level of reasoning is very low, i.e., machine code level. At this level, registers and memory are untyped, and worse still, there is no differentiation between data values and address values (pointers). A large portion of each proof typically re-establishes typing of memory locations, for example, distinguishing Integers from Booleans and from pointers.

Our approach creates a new *hybrid solution between virtual machines and proof-carrying code*, one that makes it practical to build whole operating systems on top. By raising the semantic level of the language that the proofs reason about, the proofs can become much smaller. Facts that previously required confirmation by way of proof now can be handled by axioms. Our goal is to define such a higher semantic level that is

at once effective at supporting proof-carrying code in this manner, and that can also be translated efficiently into highly performing native code on continuously evolving hardware.

As an example of what we mean by “higher semantic level”, imagine a virtual machine that supports the concept of *tagged memory*, areas of memory that have a tag stored at an offset from the pointer that is unreachable via the regular memory access instructions. The virtual machine guarantees this property, i.e., the regular memory access instructions can access only locations that lie *within the data area* of a memory block—accesses are verified to lie within the range (beginning of block, end of block), which doesn’t include the tag. Conversely, access to the tag area requires one of two *privileged instructions*, only one of which reads and the other of which writes the tag value. Such an architecture greatly simplifies certain proofs: any fragment of code that doesn’t use the privileged “write-to-tag” instruction cannot have changed the tag. Since at the higher level the tag relates to the (dynamic) type of a memory object, that implies that the type has remained constant. Effectively, this design allows us to move user-level dynamic typing and garbage collection outside of the trusted code base while maintaining efficient support of memory safety.

Conversely, traditional virtual machines need to support an unlimited number of user-defined types directly at the VM level—this implies exposing to the virtual machine not only the concept of a memory tag, but also its complete semantics. As a consequence, the garbage collector needs to become part of the trusted computing base. In our architecture, the virtual machine has only a *finite* set of data types, but supplies sufficiently powerful “hooks” for efficiently supporting an unlimited number of such types at the next higher level.

Our ultimate aim is to create a safe code platform at which proof carrying code and dynamic translation can meet effectively. Hence, we also need to demonstrate the second half of the equation: how to make such a platform efficiently implementable. The key here is our use of type separation and referentially-safe encodings on one hand (as previously demonstrated in the DARPA-sponsored safeTSA project [ADvRF01]), and of an intricate memory addressing scheme on the other hand.

Hence, a central element of our architecture is a division of concerns between the proof-carrying code mechanism and the virtual machine layer: The virtual machine layer is designed in a way to reduce the burden of proof by providing a number of inherently safe operations [FCG⁺03]. As such safe operations often involve a runtime overhead, the safe vs. the non-safe properties of the VM have to be carefully balanced. This results in including only those safe-by-construction mechanisms that have no or very little overhead compared to equivalent non-safe operations. The proof-carrying code mechanism only has to provide proof for the safety of the remaining parts of the VM architecture (Figure 2).

5 Making It Long-Term Stable

Over the past 20 years, microprocessors have undergone dramatic changes. While 20 years might appear to be a long time in terms of academic research, industrial and military applications are often designed for a lifetime extending well beyond 20 years

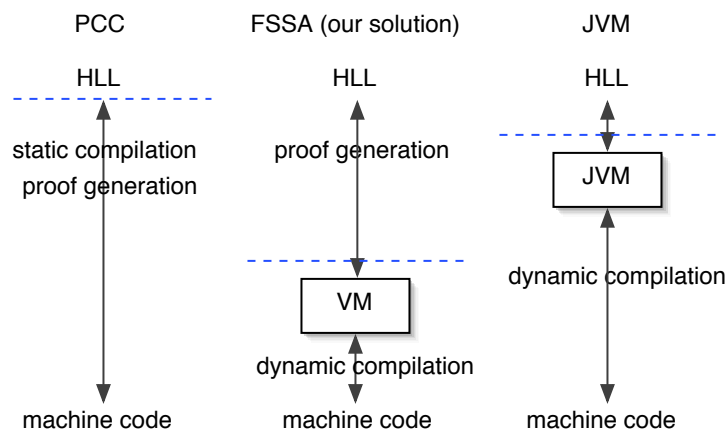


Fig. 2. Everything above the dashed line is machine independent. Compared to current PCC implementations, our framework requires shorter proofs and is machine independent. Compared to current virtual machines, dynamic compilation is simpler because the semantic distance to actual target machines is smaller. Also, we are able to perform more optimizations ahead of time.

[Age02]. To counteract obsolescence and to reduce maintenance costs of such systems, it is essential to provide a notion of stability for the executable code format. Binary compatibility at the machine code level is not necessarily the best solution in such a setting, because backward compatibility eventually leads either to a slow emulation of an old architecture on a new one (be it in hardware or software), or it necessitates binary translation—witness the large amounts of existing 8 bit and 16 bit code that is now being migrated to 32 bit processors. Obviously in this situation, both the emulator and/or the binary translator then become part of the trusted code base, potentially undermining security.

Hence, one of the key properties of our design is to hide the concrete target architecture by providing an abstract target architecture in form of a virtual machine. This virtual machine needs to be chosen so that it can provide efficient translation to several generations of actual instruction set architectures.

While existing virtual machines have predominantly used stack-based intermediate representations (Java [LY99], CLR [MG01]), our architecture uses typed registers for scalar values, reconciling efficiency and type-safety. In contrast to heavyweight VM approaches, our Virtual Machine has a finite set of types and does not directly deal with complex, composed types. However, to simplify the proof generation process, the memory access primitives make certain memory safety guarantees, on top of which type safety proofs can be easily built.

6 Comparison With Existing Approaches

The *language-based* approach to security [Koz99,SMH01] leverages program analysis and program rewriting to enforce security policies. Recent and promising exam-

ples of this approach include proof-carrying code [Nec97,NL98] (mentioned above), typed assembly language (“TAL”) [MWCG99,MCGW98], inlined execution monitors [ES99,Sch00], and information flow type systems [Mye99].

These techniques fall into two major categories, namely program rewriting and program analysis. Program analysis covers a variety of techniques that statically try to check a program’s conformance to a security policy. The primary examples of this are type-safe programming and type-based approaches to security such as typed assembly language [MWCG99,MCGW98]. Program rewriting is a complementary set of techniques that aim to enforce security by rewriting programs to conform to a security policy. Inlining security monitors [ES99] is an example of this class of techniques. An *execution monitor* (EM) monitors the execution of a *target system* and halts that system whenever it is about to violate some security policy of concern [ES99]. EMs include security kernels, reference monitors, firewalls, and most other operating system and hardware-based enforcement.

Schneider [Sch00] gives a formal characterization of the class of security policies enforceable by execution monitoring, *EM-enforceable policies*. For example, access control is EM-enforceable, while information flow and availability are not (in general).

As pointed out in [SMH01], 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 the language-based approach to security is that it is flexible and can easily express fine-grained security policies. This is in stark contrast to the inflexible, coarse grained security provided by modern operating systems.

Meta-compilation [ECCH00] is a static technique that can automatically check conformance to certain programmer-specified high-level properties such as “every lock acquired must be released”, or “these operations must be done in a certain order”. This is, however, not a language-level mechanism and statically checks existing programs. DeLine et al [DF01] have elevated a similar technique to the *language level* by using types. Their language, Vault, extends C with types which keep track of resource usage and other high level properties. Vault can be used to write low-level device drivers in manner which makes it possible to check high level security properties by virtue of them being at the language level and getting automatically checked by the compiler. For example, opening a network socket and making a connection on it involves a number of library calls that must be done in a certain order. Current languages have no mechanism to check if that order is adhered to. In Vault, this can be specified and checked by the compiler by using types.

7 Conclusion

It has been decades since type-safe programming concepts were first introduced by the titans of our field, Dijkstra, Hoare, and Wirth. Commercial software developers have largely ignored these insights—and we are all suffering the consequences today. But now, growing safety concerns in a globally networked universe are presenting compelling new arguments in favor of type-safe programming. These are highly likely to bring about a renaissance of the ideas that the JMLC conference series stands for.

Acknowledgement

This research is sponsored by the National Science Foundation under grants CCR-TC-0209163, by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), or any other agency of the U.S. Government.

I am grateful to my students Deepak Chandra, Andreas Gal, Vivek Haldar, and Vasanth Venkatachalam for many fruitful discussions about this material.

References

- [ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 1994. *SIGPLAN Notices* 29(6), June 1994.
- [ADF⁺01] Wolfram Amme, Niall Dalton, Peter H. Frohlich, Vivek Haldar, Peter S. Hous el, Jeffrey von Ronn, Christian H. Stork, Sergiy Zhenochin, and Michael Franz. Project transPROse : Reconciling Mobile-Code Security with Execution Efficiency. In *DARPA Information Survivability Conference and Exposition*, June 2001.
- [ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on sta tic single assignment form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation [PLD01]*, pages 137–147. *SIGPLAN Notices*, 36(5), May 2001.
- [Age02] Defense Advanced Research Project Agency. Program Composition of Embedded Systems (PCES), SOL BAA 02-25. <http://www.darpa.mil/baa/baa02-25.htm>, 2002.
- [BSD] BSD. Trusted BSD Project <http://www.trustedbsd.org/>.
- [cer01] CERT Advisories. <http://www.cert.org/advisories>, 2001.
- [CPM⁺98] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beat-tie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.
- [Deb] Debian. Trusted Debian Project <http://www.trusteddebian.org/>.
- [DF01] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation [PLD01]*, pages 59–69. *SIGPLAN Notices*, 36(5), May 2001.
- [ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth 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 October 2000.

- [ES99] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 1999. ACM SIGSAC, ACM Press.
- [FCG⁺03] Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermin Reig, and Ning Wang. A portable virtual machine target for proof-carrying code. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME03)*, San Diego, California, June 2003.
- [HSF02] Vivek Haldar, Christian H. Stork, and Michael Franz. The Source is the Proof. In *New Security Paradigms Workshop*, Sep 2002.
- [Koz99] Dexter Kozen. Language-Based Security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.
- [MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-Based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.
- [MG01] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. [http://research.microsoft.com/~emeijer/papers/clr.pdf](http://research.microsoft.com/~emeijer/papers clr.pdf), 2001.
- [MPS⁺03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The Spread of the Sapphire/Slammer Worm. <http://www.silicondefense.com/research/sapphire/>, 2003.
- [MWC99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
- [Mye99] Andrew 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*, pages 228–241, San Antonio, Texas, January 20–22, 1999.
- [Nec97] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.
- [NL98] George C. Necula and Peter Lee. Safe, Untrusted Agents using Proof-Carrying Code. In G. Vigna, editor, *Safe, Untrusted Agents using Proof-Carrying Code*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, Berlin, Germany, 1998.
- [NMW02] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages*, 2002.
- [NSA] National Security Agency. Security Enhanced Linux <http://www.nsa.gov/selinux/>.
- [PLD01] *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.
- [Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [SMH01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A Language-Based Approach to Security. In *Informatics*, pages 86–101, 2001.
- [WFBA00] David Wagner, Jeffrey S. Foster, Eric 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, 2000.