The Halting Problems of Network Stack Insecurity

LEN SASSAMAN, MEREDITH L. PATTERSON, SERGEY BRATUS, AND ANNA SHUBINA

Len Sassaman was a PhD student in the COSIC research group at Katholieke Universiteit Leuven. His early work with the Cypherpunks on the Mixmaster anonymous remailer system and the Tor Project helped establish the field of anonymity research, and in 2009 he and Meredith began formalizing the foundations of language-theoretic security, which he was working on at the time of his death in July 2011. He was 31.

Meredith L. Patterson is a software engineer at Red Lambda. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium.
mlp@thesmartpolitenerd.com

Sergey Bratus is a Research Assistant Professor of Computer Science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in Mathematics from Northeastern University and worked at BBN Technologies on natural language processing research before coming to Dartmouth.
sergey@cs.dartmouth.edu

Anna Shubina chose "Privacy" as the topic of her doctoral thesis and was the operator of Dartmouth's Tor exit node when the Tor network had about 30 nodes total. She is currently a research associate at the Dartmouth Institute for Security, Technology, and Society, and manages the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research.
ashubina@cs.dartmouth.edu

Everyday computer insecurity has only gotten worse, even after many years of concerted effort. We must be missing some fundamental yet easily applicable insights into why some designs cannot be secured, how to avoid investing in them and re-creating them, and why some result in less insecurity than others. We posit that by treating valid or expected inputs to programs and network protocol stacks as *input languages that must be simple to parse* we can immensely improve security. We posit that the opposite is also true: *a system whose valid or expected inputs cannot be simply parsed cannot in practice be made secure.*

In this article we demonstrate why we believe this a defining issue and suggest guidelines for designing protocols as secure input languages—and, thus, secure programs. In doing so, we link the formal languages theory with experiences and intuitions of both program exploitation and secure programming.

Indeed, a system's security is largely defined by what computations can and cannot occur in it under all possible inputs. Parts of the system where the input-driven computation occurs are typically meant to act together as a *recognizer* for the inputs' validity (i.e., they are expected to reject bad inputs). Exploitation—an *unexpected* input-driven computation—usually occurs there as well; thinking of it as an input language *recognizer bug* helps find it (as we will show).

Crucially, for complex inputs (input languages) the recognition that matches the programmer's expectations can be equivalent to the "halting problem"—that is, UNDECIDABLE. Then no generic algorithm to establish the inputs' validity is

SIDEBAR

The Chomsky hierarchy ranks languages according to their expressive power in a strict classification of language/grammar/automata classes that establishes a correspondence between language classes, their grammars, and the minimum strength of a computational model required to recognize and parse them.

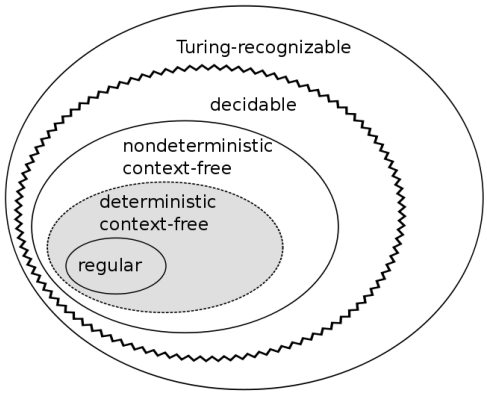
Regular languages, the weakest class of languages, need only a finite state machine and can be parsed with regular expressions.

Unambiguous context-free grammars, the first class that allows some recursively nested data structures, need deterministic pushdown automata (i.e., they require adding a stack to the limited memory of a finite state machine).

Ambiguous context-free grammars need nondeterministic pushdown automata to account for ambiguity.

The more powerful classes of languages, *context-sensitive languages* and *recursively enumerable languages*, require linear bounded automata and Turing machines, respectively, to recognize them. Turing-recognizable languages are UNDECIDABLE. There is a boundary of decidability which it is unwise for an input language or protocol designer to cross, as is discussed in Principle 1 (p. 29, below).

For the regular and deterministic context-free grammars, the *equivalence problem*—do two grammars produce exactly the same language?—is DECIDABLE. For all other classes of grammar, the equivalence problem is UNDECIDABLE, and they should be avoided wherever security relies on computational equivalence of parser implementations, as Principle 2 posits.



possible, no matter how much effort is put into making the input data "safe." In such situations, whatever actual checks the software performs on its inputs at various points are unlikely to correspond to the programmer assumptions of validity or safety at these points or after them. This greatly raises the likelihood of exploitable input handling errors.

A protocol that appears to frustratingly resist efforts to implement it securely (or even to watch it effectively with an IDS) behaves that way, we argue, because its very design puts programmers in the position of unwittingly trying to solve (or approximate a solution to) an UNDECIDABLE problem. Conversely, understanding the flavor of mismatch between the expected and the required (or impossible) recognizer power for the protocol as an input language to a program eases the task of 0-day hunting.

Yet we realize it is all too easy to offer general theories of insecurity without improving anything in practice. We set the following three-pronged practical test as a threshold for a theory's usefulness, and hope to convince the reader that ours passes it. We posit that a theory of insecurity must:

- explain why designs that are known to practitioners as hard to secure are so, by providing a fundamental theoretical reason for this hardness;
- give programmers and architects clear ways to avoid such designs in the future, and prevent them from misinvesting their effort into trying to secure unsecurable systems rather than replacing them;
- significantly facilitate finding insecurity when applied to analysis of existing systems and protocols—that is, either help point out new classes of 0-day vulnerabilities or find previously missed clusters of familiar ones.

As with any attempted concise formulation of a general principle, parts of an up-front formulation may sound similar to some previously mooted pieces of security wisdom; to offset such confusion, we precede the general principles with a number of fundamental examples. We regret that we cannot review the large corpus of formal methods work that relates to various aspects of our discussion; for this, we refer the reader to our upcoming publications (see langsec.org).

The Need for a New Understanding of Computer (In)Security

Just as usefulness of a computing system and its software in particular has become synonymous with it being network-capable, network-accessible, or containing a network stack of its own, we are clearly at an impasse as to how to combine this usefulness with security.

A quote commonly attributed to Einstein is, "The significant problems we face cannot be solved at the same level of thinking we were at when we created them." We possess sophisticated taxonomies of vulnerabilities and, thanks to hacker research publications, intimate knowledge of how they are exploited. We also possess books on how to program

securely, defensively, and robustly. Yet for all this accumulated knowledge and effort, insecurity prevails—a sure sign that we are still missing this "next level" in theoretical understanding of how it arises and how to control it.

A Language Theory Look at Exploits and Their Targets

The "common denominator" of insecurity is *unexpected computation* (a.k.a. "malicious computation") reliably caused by *crafted inputs* in the targeted computing environment—to the chagrin of its designers, implementers, and operators. As we point out in [2], this has long been the intuition among hacker researchers, leading them to develop a sophisticated approach that should inform our theoretical next step.

The exploit is really a *program* that executes on a collection of the target's computational artifacts, including bugs such as memory corruptions or regular features borrowed for causing unexpected control or data flows. The view of creating exploits as a kind of macro-assembler programming with such artifacts as "primitives" or macros has firmly established itself (e.g., [3]), the full collection of such artifacts referred to as a "weird machine" within the target. In these terms, "malicious computation" executes on the "weird machine," and, vice versa, the weird machine is what runs the exploit program.

The crucial observation is that the exploit program, whatever else it is, is expressed as *crafted input* and is processed by the target's *input processing routines*. Furthermore, it is these processing routines that either provide the bugs for the weird machine's artifacts or allow crafted input clauses to make their way to such artifacts further inside the target.

Thus a principled way to study crafted input exploit programs and targets in conjunction is to study both the totality of the target's intended or accepted inputs as a language in the sense of the formal language theory, and the input-handling routines as machines that recognize this language. This means, in turn, that evaluating the design of the program's input-handling units and the program itself based on the properties of these languages is indispensable to security analysis.

Throughout this article, we refer to a program's inputs and protocols *interchangeably*, to stress that we view protocols as sequences of inputs, which for every intended or accepted protocol exchange or conversation should be considered as a part of the respective input language. Moreover, we speak of applications' inputs and network stack inputs interchangeably, as both stack layers and applications contain input-handling units that form an important part of the overall system's attack surface.

Let us now apply this general principle to the study of insecurities and finding 0-days in network stacks. We will then explain how it quantifies the hardness of secure design and testing, and helps the designers steer around potentially unsecurable or hard-to-secure designs.

We note that although insecurity obviously does not stop at input handling (which our own examples of composition-based insecurity will illustrate), a provably correct input parser will greatly reduce the reachable attack surface—even though it lacks the magical power to make the system unexploitable. Moreover, the language-theoretic approach applies beyond mere input-parsing, as it sheds light on such questions as, "Can a browser comprehensively block 'unsafe' JavaScript for some reasonable security model and what computational power would be required to do so?" and "Does JSON promote safer Web app development?"

Language-theoretic Attacks on Protocol Parsers

Since the 1960s, programming-language designers have employed automated *parser generators* to translate the unique defining grammar of a machine-parsable language into a parser for that language. Every parser for a given language or protocol is also a *recognizer* for that language: it accepts strings (e.g., binary byte strings) that are valid in its language, and rejects invalid ones—and is therefore a security-crucial component. Although this approach has been of great benefit to compiler and interpreter design, it has largely gone unused with respect to *protocol design and implementation*—at great detriment to security.

Most protocol implementations, in particular network protocol stacks, are still built on essentially *handwritten recognizers*. This leads to implementation errors that introduce security holes or actually accept a broader set of strings than the protocol recognizes. This, in turn, propagates security problems into other implementations that need to accommodate the broken implementation (e.g., several Web servers incorrectly implement TLS/SSL 3.0 in order to interoperate with Internet Explorer [9]).

Furthermore, most approaches to *input validation* also employ handwritten recognizers, at most using regular expressions to whitelist acceptable inputs and/or blacklist potentially malicious ones. Such recognizers, however, are powerless to validate stronger classes of languages allowing for recursively nested data structures, such as context-free languages, which require more powerful recognizers. The sidebar (p. 23, above) describes the Chomsky hierarchy of language grammar classes and their respective recognizer automata classes, by the required computational strength.

This suggests that our language-theoretic approach should reveal clusters of potential 0-days in network stacks, starting at the top, and descending through the middle layers to its very bottom, the PHY layer. Indeed, consider the following examples.

X.509 Parsing

In [8], Kaminsky, Patterson, and Sassaman observed that ASN.1 requires a context-sensitive parser, but the specification of ASN.1 is not written in a way conducive to implementing a parser generator, causing ASN.1 parsers to be handwritten. The parse trees generated by these parsers would thus most likely be different, and their mismatches would indicate potential vulnerabilities.

The authors examined how different ASN.1 parsers handle X.509 documents, focusing on unusual representations of their components, such as Common Name. The results of this examination were numerous vulnerabilities, some of which, when exploited, would allow an attacker to claim a certificate of any site.

Here are just two examples of the many problems with X.509 they discovered using this method:

1. Multiple Common Names in one X.509 Name are handled differently by different implementations. The string `CN=www.badguy.com/CN=www.bank.com/CN=www.bank2.com/CN=*` will pass validation by OpenSSL, which returns only

the first Common Name, but authenticate both www.bank.com and www.bank2.com for Internet Explorer, and authenticate all possible names in Firefox.

2. Null terminators in the middle of an X.509 Name can cause some APIs to see different names than others. In case of the name "www.bank.com00.badguy.com," some APIs would see "badguy.com," but IE's CryptoAPI and Firefox's NSS will see "www.bank.com". Due to NSS's permissive parsing of wildcards, it would also accept a certificate for "*00.badguy.com" for all possible names.

It should be stressed that individual protocol parser vulnerabilities can be found in other ways; for instance, the second item above was independently discovered by Moxie Marlinspike. However, by themselves they may look like "random" bugs and show neither the size of the attack surface nor the systematic nature of the implementers' errors, whereas a language-theoretic analysis reveals the roots of the problem; the difference is that between finding a nugget and striking a gold mine of 0-days.

SQL Parsing and Validation

In [7], Hansen and Patterson discuss SQL injection attacks against database applications. SQL injection attacks have been extremely successful, due to both the complicated syntax of SQL and application developers' habit of sanitizing SQL inputs by using regular expressions to ban undesirable inputs, whereas regular expressions are not powerful enough to validate non-regular languages.

In particular, SQL was context-free until the introduction of the WITH RECURSIVE clause, at which point it became Turing-complete [4] (although in some SQL dialects it may also be possible to concoct a Turing machine using triggers and rules; we are indebted to David Fetter for this observation). Mere regular expressions, which recognize a weaker class of languages, could not validate (i.e., recognize) it even when it was context-free. Turing completeness makes validation hopeless, since recognizing such languages is an undecidable problem. Trying to solve it in all generality is a misinvestment of effort.

The authors suggest that a correct way to protect from SQL injection is to define a safe subset of SQL, which is likely to be a very simple language for any particular application accepting user inputs, and to proceed by generating a parser for that language. This approach offers complete security from SQL injection attacks.

Generalization: Parse Tree Differential Analysis

In [8] Kaminsky, Sassaman, and Patterson further generalized their analysis technique to arbitrary protocols, developing *the parse tree differential attack*, a powerful technique for discovering vulnerabilities in protocol implementations, generating clusters of 0-days, and saving effort from being misinvested into incorrect solutions. This attack compares parse trees corresponding to two different implementations of the same protocol. Any differences in the parse trees indicate potential problems, as they demonstrate the existence of inputs that will be parsed differently by the two implementations.

This method applies everywhere where structured data is marshalled into a string of bytes and passed to another program unit, local or remote. In particular, it should be a required part of security analysis for any distributed system's design. We will discuss its further implications for *secure composition* below.

IDS Evasion and Network Stack Fingerprinting

Differences in protocol parsing at Layers 3 and 4 of TCP/IP network stacks have long been exploited for their fingerprinting (by *Nmap*, *Xprobe*, etc.). Then it was discovered that the impact of these differences on security was much stronger than just enabling reconnaissance: network streams could be crafted in ways that made the NIDS or "smart" firewalls "see" (i.e., have its network stack reassemble) completely different session contents than the targets they protected.

The seminal 1998 paper by Ptacek and Newsham [10] was the first to broach this new research direction. A lot of work followed; for a brief summary see [11]. In retrospect, Ptacek and Newsham's paper was a perfect example of analysis that implicitly treated network protocol stacks' code as protocol recognizers. It also suggested that the target and the NIDS were parts of a *composed* system, and a NIDS's security contribution was ad hoc at best (and negative at worst, for creating a false expectation of security) unless it matched the target in this composition.

Digital Radio Signaling

Recent discovery of overlooked signaling issues as deep as the PHY layer of a broad range of digital radio protocols (802.15.4, Bluetooth, older 802.11, and other popular RF standards) [6] shows another example of a surprisingly vulnerable design that might have gone differently had a language-theoretic approach been applied from the start—and that language-theoretic intuitions have helped to uncover.

The authors demonstrated that the abstraction of PHY layer encapsulation of Link Layer frames in most forms of unencrypted variable-frame-length digital radio can be violated simply by ambient noise. In particular, should the preamble or Start of Frame Delimiter (SFD) be damaged, the "internal" bytes of a frame (belonging to a crafted higher layer protocol payload) can be received by local radios as a PHY layer frame. This essentially enables remote attackers who can affect payloads at Layer 3 and above on the local RF to inject malicious frames without ever owning a radio.

This is certainly not what most Layer 2 and above protocol engineers expect of these PHY layer implementations. From the language recognizer standpoint, however, it is obvious that the simple finite automaton used to match the SFD in the stream of radio symbols and so distinguish between the noise, signaling, and payloads can be easily tricked into "recognizing" signaling as data and vice versa.

Defensive Recognizers and Protocols

To complete our outlook, we must point to several successful examples of program and protocol design that we see as proceeding from and fulfilling related intuitions.

The most recent and effective example of software specifically designed to address the security risks of an input language in common Internet use is Blitzableiter by Felix 'FX' Lindner and Recurity Labs [12, 13]. It takes on the task of *safely* recognizing Flash, arguably the most complex input language in common Internet use, due to two versions of bytecode allowed for backward compatibility and the complex SWF file format; predictably, Flash is a top exploitation vector with continually surfacing vulnerabilities. Blitzableiter (a pun on lightning rod) is an armored recognizer for Flash, engineered to maximally suppress implicit data and control flows that help turn ordinary Flash parsers into "weird machines."

Another interesting example is the observations by D.J. Bernstein on the 10 years of qmail [13]. We find several momentous insights in these, in particular *avoiding parsing* (i.e., in our terms, dealing with non-trivial input languages) whenever possible as a way of making progress in eliminating insecurity, and pointing to handcrafting input-handling code for efficiency as a dangerous distraction. In addition, Bernstein stresses using UNIX context isolation primitives as a way of enforcing *explicit data flows* (in our terms, hobbling construction of "weird machines"). Interestingly, Bernstein also names the Least Privilege Principle—as currently understood—as a distraction; we argue that this principle needs to be updated rather than discarded, and we see Bernstein's insights as being actually in line with our proposed update (see below).

There are also multiple examples of protocols designed with easy and unambiguous parsing in mind. Lacking space for a comprehensive review of the protocol design space, we point the reader to our upcoming publication, and only list a few examples here:

- The ATM packet format is a regular language, the class of input languages parsable with a finite-state machine, easiest to parse, which helps avoid signaling attacks as discussed above. The same is true for other fixed-length formats.
- JSON is arguably the closest to our recommendation for a higher-layer language for encoding and exchanging complex, recursive objects between parts of a distributed program. Such a language needs to be context-free (the classic example of this class is S-expressions), but not stronger.

Language-theoretic Principles of Secure Design

Decidability matters. Formally speaking, a correct protocol implementation is defined by the decision problem of whether the byte string received by the stack's input handling units is a member of the protocol's language. This problem has two components: first, whether the input is syntactically valid according to the grammar that specifies the protocol, and second, whether the input, once recognized, generates a valid state transition in the state machine that represents the logic of the protocol. The first component corresponds to the parser and the second to the remainder of the implementation.

The difficulty of this problem is directly defined by the class of languages to which the protocol belongs. Good protocol designers don't let their protocols grow up to be Turing-complete, because then the decision problem is UNDECIDABLE.

In practice, undecidability suggests that *no amount of programmer or QA effort is likely to expose a comprehensive selection of the protocol's exploitable vulnerabilities related to incorrect input data validity assumptions.* Indeed, if no generic algorithm to establish input validity is possible, then whatever actual validity checks the software performs on its inputs at various points are unlikely to correspond to the programmer's assumptions of such validity. Inasmuch as the target's potential vulnerability set is created by such incorrect assumptions, it is likely to be large and non-trivial to explore and prune.

From malicious computation as the basis of the threat model and the language-theoretic understanding of inputs as languages, several bedrock security principles follow:

Principle 1: Starve the Turing Beast—Request and Grant Minimal Computational Power

Computational power is an important and heretofore neglected dimension of the attack surface. Avoid exposing unnecessary computational power to the attacker.

An input language should only be as computationally complex as absolutely needed, so that the computational power of the parser necessary for it can be minimized. For example, if recursive data structures are not needed, they should not be specified in the input language.

The parser should be no more computationally powerful than it needs to be. For example, if the input language is context-free, then the parser should be no more powerful than a deterministic pushdown automaton.

For Internet engineers, this principle can be expressed as follows:

- a parser must not provide more than the minimal computational strength necessary to interpret the protocol it is intended to parse;
- protocols should be designed to require the computationally weakest parser necessary to achieve the intended operation.

An implementation of a protocol that exceeds the computational requirements for parsing that protocol's inputs should be considered broken.

Protocol designers should design their protocols to be as weak as possible. Any increase in computational strength of input should be regarded as a grant of additional privilege, thus increasing security risk. Such increases should therefore be entered into reluctantly, with eyes open, and should be considered as part of a formal risk assessment. At the very least, the designer should be guided by the Chomsky hierarchy (described in the sidebar, p. 23).

Input-handling parts of most programs are essentially Turing machines, whether this level of computational power is needed or not. From the previously discussed *malicious computation* perspective of exploitation it follows that this delivers the full power of a Turing-complete environment into the hands of the attacker, who finds a way of leveraging it through crafted inputs.

Viewed from the venerable perspective of Least Privilege, Principle 1 states that *computational power is privilege, and should be given as sparingly as any other kind of privilege to reduce the attack surface.* We call this extension the *Minimal Computational Power Principle.*

We note that recent developments in common protocols run contrary to these principles. In our opinion, this heralds a bumpy road ahead. In particular, HTML5 is Turing-complete, whereas HTML4 was not.

Principle 2: Secure Composition Requires Parser Computational Equivalence

Composition is and will remain the principal tool of software engineering. Any principle that aims to address software insecurity must pass the test of being applicable to practical software composition, lest it forever remain merely theory. In particular, it should specify how to maintain security in the face of (inevitable) composition—including, but not limited to, distributed systems, use of libraries, and lower-layer APIs.

From our language-theoretic point of view, any composition that involves converting data structures to streams of bytes and back for communications between components necessarily relies for its security on the different components of the system performing *equivalent* computations on the input languages.

However, computational equivalence of automata/machines accepting a language is a highly non-trivial language-theoretic problem that becomes UNDECIDABLE starting from non-deterministic context-free languages (cf. the sidebar for the decidability of the equivalence problem).

The X.509 example above shows that this problem is directly related to insecurity of distributed systems' tasks. Moreover, undecidability essentially precludes construction of efficient code testing and/or verification algorithmic techniques and tools.

On the Relevance of Postel's Law

This leads to a re-evaluation of Postel's Law and puts Dan Geer's observations in "Vulnerable Compliance" [5] in solid theoretical perspective.

Postel's *Robustness Principle* (RFC 793), best known today as *Postel's Law*, laid the foundation for an interoperable Internet ecosystem. In his specification of TCP, Postel advises to "be conservative in what you do, be liberal in what you accept from others." Despite being a description of the principle followed by TCP, this advice became widely accepted in IETF and general Internet and software engineering communities as a core principle of protocol implementation.

However, this policy maximizes interoperability at the unfortunate expense of consistent parser behavior, and thus at the expense of security.

Why Secure Composition Is Hard

The second principle provides a powerful theoretical example of why *composition*—the developer's and engineer's primary strategy against complexity—is hard to do securely. Specifically, a composition of communicating program units must rely on *computational equivalence* of its input-handling routines for security (or even correctness when defined); yet such equivalence is UNDECIDABLE for complex protocols (starting with those needing a nondeterministic pushdown automaton as a recognizer of their input language), and therefore cannot in practice be checked even for differing implementations of the same communication logic.

Conversely, this suggests a principled approach for reducing insecurity of composition: keep the language of the messages exchanged by the components of a system to a necessary minimum of computational power required for their recognition.

Parallels with Least Privilege Principle

The understanding of "malicious computation" programmed by crafted inputs on the "weird machine" made of a target's artifacts as a threat naturally complements and extends the Least Privilege Principle as a means of containing the attacker. In particular, just as the attacker should not be able to spread the compromise beyond the vulnerable unit or module, he should not be able to propagate it beyond the minimal computational power needed. This would curtail his ability to perform malicious computations.

Thus the Least Privilege Principle should be complemented by the Minimal Computational Power Principle. This approach should be followed all the way from the application protocol to hardware. In fact, we envision *hardware* that limits itself from its current Turing machine form to weaker computational models according to the protocol parsing tasks it must perform, lending no more power to the parsing task than the corresponding language class requires—and therefore no more power for the attacker to borrow for exploit programs in case of accidental exposure, starving the potential "weird machines" of such borrowed power. This restriction can be accomplished by reprogramming the FPGA to only provide the appropriate computational model—say, finite automaton or a pushdown automaton—to the task, with appropriate hardware-configured and enforced isolation of this environment from others (cf. [1]).

Conclusion

Computer security is often portrayed as a never-ending arms race between attackers seeking to exploit weak points in software and defenders scrambling to defend regions of an ever-shifting battlefield. We hold that the front line is, instead, a bright one: the system's security is defined by what computations can and cannot occur in it under all possible inputs. To approach security, the system must be analyzed as a recognizer for the language of its valid inputs, which must be clearly defined by designers and understood by developers.

The computational power required to recognize the system's valid input language(s) must be kept at a minimum when designing protocols. This will serve to both reduce the power the attacker will be able to borrow, and help to check that handling of structured data across the system's communicating components is *computationally equivalent*. The lack of such equivalence is a core cause of insecurity in network stacks and in other composed and distributed systems; undecidability of checking such equivalence for computationally demanding (or ambiguously specified) protocols is what makes securing composed systems hard or impossible in both theory and practice.

We state simple and understandable but theoretically fundamental principles that could make protection from unexpected computations a reality, if followed in the design of protocols and systems. Furthermore, we suggest that in future designs hardware protections should be put in place to control and prevent exposure of unnecessary computational power to attackers.

References

[1] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith, "New Directions for Hardware-Assisted Trusted Computing Policies" (position paper), 2008.

[2] Sergey Bratus, Michael Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina, "Exploit Programming: From Buffer Overflows to Theory of Computation," in preparation.

[3] Thomas Dullien, "Exploitation and State Machines: Programming the 'Weird Machine,' Revisited," Infiltrate Conference, April 2011: http://www.immunityinc.com/infiltrate/presentations/Fundamentals_of_exploitation_revisited.pdf.

[4] David Fetter, "Lists and Recursion and Trees, Oh My!" OSCON, 2009.

[5] Dan Geer, "Vulnerable Compliance," *;login:*, vol. 35, no. 6 (December 2010): http://www.usenix.org/publications/login/2010-12/pdfs/geer.pdf.

[6] Travis Goodspeed, Sergey Bratus, Ricky Melgares, Rebecca Shapiro, and Ryan Speers, "Packets in Packets: Orson Welles' In-Band Signaling Attacks for Modern Radios," 5th USENIX Workshop on Offensive Technologies, August 2011: http://www.usenix.org/events/woot11/tech/final_files/Goodspeed.pdf.

[7] Robert J. Hansen and Meredith L. Patterson, "Guns and Butter: Towards Formal Axioms of Input Validation," Black Hat USA, August 2005: http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Hansen-Patterson/HP2005.pdf.

[8] Dan Kaminsky, Len Sassaman, and Meredith Patterson, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure," Black Hat USA, August 2009: http://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[9] Katsuhiko Momoi, "Notes on TLS-SSL 3.0 Intolerant Servers": http://developer.mozilla.org/en/docs/Notes_on_TLS_-_SSL_3.0_Intolerant_Servers, 2003.

[10] Thomas H. Ptacek and Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection, " technical report, Secure Networks, Inc., January 1998: http://insecure.org/stf/secnet_ids/secnet_ids.html.

[11] Sumit Siddharth, "Evading NIDS, Revisited": http://www.symantec.com/connect/articles/evading-nids-revisited.

[12] Stefan Krempl, "Protection against Flash Security Holes," December 30, 2009: http://www.h-online.com/security/news/item/26C3-Protection-against-Flash-security-holes-893689.html.

[13] Felix "FX" Lindner, "The Compromised Observer Effect," *McAfee Security Journal*, 6 (2010): 16–19.

[13] Daniel J. Bernstein, "Some Thoughts on Security after 10 Years of qmail 1.0," November 1, 2007: cr.yp.to/qmail/qmailsec-20071101.pdf.