Security Applications of Formal Language Theory

Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto

***Abstract*—We present a formal language theory approach to improving the security aspects of protocol design and message-based interactions in complex composed systems. We argue that these aspects are responsible for a large share of modern computing systems' insecurity. We show how our approach leads to advances in input validation, security modeling, attack surface reduction, and ultimately, software design and programming methodology. We cite examples based on real-world security flaws in common protocols, representing different classes of protocol complexity. We also introduce a formalization of an exploit development technique, the parse tree differential attack, made possible by our conception of the role of formal grammars in security. We also discuss the negative impact unnecessarily increased protocol complexity has on security. This paper provides a foundation for designing verifiable critical implementation components with considerably less burden to developers than is offered by the current state of the art. In addition, it offers a rich basis for further exploration in the areas of offensive analysis and, conversely, automated defense tools, and techniques.**

***Index Terms*—Language-theoretic security, secure composition, secure protocol design.**

I. Introduction

Composition is the primary engineering means of complex system construction. No matter what other engineering approaches or design patterns are applied, the economic reality is that a complex computing system will ultimately be pulled together from components made by different people and groups of people. The interactions between these components imply creation of the components' communication boundaries and usher in messaging protocols. These boundaries and protocols then become attack surfaces, dominating the current landscape of Internet insecurity.

For the traditional division of a system into hardware, firmware, and software, and of software into device drivers, generic OS kernel and its sublayers, and various application software stacks and libraries, the fact of this composition is so obvious that it is commonly dismissed as trivial. How else can one build modern computers and modern software if not in a modular way? Moreover, modularity is supposed to be good for security and reliability because without them programming would be intractable.

However, composing communicating components securely has emerged as the primary challenge to secure system construction. Security practitioners know that communication boundaries become attack targets of choice, and that vulnerabilities are often caused by unexpected interactions across components. Yet, the reasons for this are elusive. Attackers naturally desire reliable execution of their exploits, which leads them to target communication boundaries as the best-described parts of components, with tractable state. Still, this does not explain our collective inability to design systems without unwanted interactions.

In this paper, we argue that to finally get it right security-wise, we need a new stronger computational-theoretic understanding of message-based interactions between components. We use formal language complexity arguments to explain why certain protocol and message format design decisions are empirically known to be wellsprings of vulnerabilities, and why the respective software components do not seem to yield to concerted industry efforts to secure them.

We show that there are strong computational-theoretic and formal language-theoretic reasons for the challenges of secure messaging-based composition, and chart basic design principles to reduce these challenges. In particular, we show that the hard challenges of safe input handling and secure communication arise due to the underlying theoretically hard or unsolvable (i.e., undecidable) problems that certain protocol designs and implementations force programmers to solve to secure them. We posit that the (unwitting) introduction of such problems in the protocol design stage explains the extreme propensity of certain protocols and message formats to yield a seemingly endless stream of 0-day vulnerabilities despite efforts to stem it, and the empirical hopelessness of fixing these protocols and message formats without a fundamental redesign.

We also chart ways to avoid designs that are prone to turning into security nightmares for future Internet protocols. Empirically, attempts to solve an engineering problem that implies a good enough (or 80%/20%) solution to the underlying undecidable theory problem are doomed to frustration and failure, which manifests in many ways such as no amount of testing sufficing to get rid of bugs, or the overwhelming complexity and not-quite-correct operation of the automation or detection tools created to deal with the problem. Thus, avoiding such problems in the first place (at the design stage) saves both misinvestment of programming effort and operational costs.

We note that many practical systems were neither designed not developed with security in mind. However, simply giving

Manuscript received October 9, 2011; revised April 17, 2012; accepted September 22, 2012. Date of current version July 3, 2013. The work of L. Sassaman was supported in part by the Research Council K. U. Leuven: GOA TENSE under Grant GOA/11/007, and by the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

L. Sassaman was with Katholieke Universiteit Leuven, Leuven Bus 5005 3000, Belgium.

M. L. Patterson is with Red Lambda, Longwood, FL 32779 USA (e-mail: mlp@upstandinghackers.com).

S. Bratus is with the Dartmouth College, Hanover, NH 03755 USA (e-mail: sergey@cs.dartmouth.edu).

M. E. Locasto is with the University of Calgary, Calgary, AB T2N 1N4, Canada (e-mail: locasto@ucalgary.ca).

Digital Object Identifier 10.1109/JSYST.2012.2222000

1932-8184/$31.00 © 2012 IEEE

up on them wholesale and rebuilding them from scratch is hardly an option. Rather than condemning such systems based on their history of exploitation, we show a path to improving them, as long as their commmunication boundaries can be identified, analyzed, and improved according to our proposed approach.

Our argument focuses on the application of fundamental decidability results to the two basic challenges of composed and distributed system construction that replies on communication between components: safely accepting and handling inputs in every component, and identical interpretation of messages passed between components at every endpoint. In particular, we consider the following two perspectives on composition.

1) *Single-component perspective:* A component in a complex system must accept inputs or messages across one or more interfaces. This creates an attack surface, leveraged by an absolute majority of exploitation techniques. We discuss hardening the attack surface of each component against malicious crafted inputs, so that a component is capable of rejecting them without losing integrity and exhibiting unexpected behavior—in short, without being exploited.
2) *Multicomponent perspective:* As components exchange messages, they must ensure that, despite possible implementation differences, they interpret the messages identically. Although this requirement appears to be trivially necessary for correct operation, in reality different implementations of a protocol by different components produce variations, or mutually intelligible dialects, with message semantic differences masked (and therefore ignored) in nonmalicious exchanges. A smaller but important class of attack techniques leverages such differences, and can lead to devastating attacks such as those on X.509 and ASN.1 discussed in this paper.

The importance of these requirements is an empirical fact of the Internet security experience (see [1]–[3]), which our paper puts in solid theory perspective.

A. Structure of This Paper

We start with the motivation of our approach in Section II and outline our case for the formal language-theoretic approach to security in view of state-of-the-art exploits and defenses. We review the necessary background formalisms in Section III.

Then, in Section IV, we explain how these general formalisms apply to exploitation of computing systems, and illustrate this application for several well-known classes of practical exploitation techniques. In doing so, we connect the corresponding classes of attacks with the formal language properties of targeted data structures, which provides a novel and definitive way to analyze various suggested defenses.

In Section V, we show how to apply formal language-theoretic techniques to achieve rigorous nonheuristic input validation. We start our discussion with SQL validation, but also show that the same approach applies to more complex languages such as PKCS#1 (in Section V-B we prove that PKCS#1 is context-sensitive). We also discuss flaws in previous validation approaches, and show why these flaws matter for practical security.

The discussion of flaws leads us to Section VI, in which we present a new technique for security analysis of differences between mutually intelligible language dialects that arise from implementation differences. This technique, parse tree differential analysis, proved a powerful tool for enhancing code auditing and protocol analysis.

In Section VII, we show that the challenges and failures of IDS/IPS, arguably the most common form of security composition, can be explained via language-theoretic computational equivalence. We conclude with an outline of future work.

II. Why Security Needs Formal Language Theory

We posit that input verification using formal language theoretic methods—whether simply verifying that an input to a protocol constitutes a valid expression in the protocol's grammar or also verifying the semantics of input transformations—is an overlooked but vital component of protocol security, particularly with respect to implementations. Simply put, a protocol implementation cannot be correct unless it recognizes input correctly, and should otherwise be considered broken.

Formal software verification seeks to prove certain safety (nothing bad happens) and liveness (something good happens, eventually) properties of program computations: if every computation a program can perform satisfies a particular property, the program is safe (or, respectively, live) with respect to that property [4]. Program verification in the general case is undecidable, and although many approaches to falsification and verification of properties have been developed, unsolved and unsolvable problems with the scalability and completeness of algorithmic verification have prevented formal correctness from displacing testing and code auditing as the industry gold standard for software quality assurance [5]. However, programs that implement protocols—that is to say, routines that operate over a well-defined input language[1]—share one characteristic that can be leveraged to dramatically reduce their attack surfaces: their input languages can—and, we posit, should—in general be made decidable and decided in a tractable fashion. We show that this requirement of being well specified and tractably decidable is in fact a crucial prerequisite of secure design and, in fact, its violation is the source of much present-day computer insecurity.

Inputs to system components such as web browsers, network stacks, cryptographic protocols, and databases are formally specified in standards documents, but by and large, implementations' input handling routines parse the languages these standards specify in an *ad hoc* fashion. Attacks such as the Bleichenbacher PKCS#1 forgery [6], [7] show what can happen when an ad hoc input-language implementation fails to provide all the properties of the input language as actually specified. In more recent work [8], we have shown that variations among implementations can be exploited to subvert the interoperation of these implementations, and that ambiguity or

[1]This includes file formats, wire formats and other encodings, and scripting languages, and the conventional meaning of the term, e.g., finite-state concurrent systems such as network and security protocols.

underspecification in a standard increases the chances of vulnerability in otherwise standards-compliant implementations.

On this basis, we argue that mutually intelligible dialects of a protocol cannot make guarantees about their operation because the problem EQUIVALENT($\mathcal{L}(G) = \mathcal{L}(H)$) is undecidable when G and H are grammars more powerful than deterministic context-free [9], [10]. We also observe that systems that consist of more than one component have inherent, de facto design contracts for how their components interact, but generally do not enforce these contracts; SQL injection attacks (hereafter SQLIA), for instance, occur when an attacker presents a database with an input query that is valid for the database in isolation, but invalid within the context of the database's role in a larger application.

Since well-specified input languages are in the main decidable[2] (or can be made so), there is no excuse for failing to verify inputs with the tools that have existed for this exact purpose for decades: formal parsers. We will examine input verification from several different angles and across multiple computability classes, highlight the unique problems that arise when different programs that interoperate over a standard permit idiosyncratic variations to that standard, and show formally how to restrict the input language of a general-purpose system component (such as a database) so that it accepts only those inputs that it is contractually obligated to accept.

Given the recent advent of provably correct, guaranteed-terminating parser combinators [16] and parser generators based on both parsing expression grammars [17] and context-free grammars (CFGs) [18], we hold that the goal of general formal parsing of inputs is within practical reach. Moreover, informal guarantees of correct input recognition are easy to obtain via commonly available libraries and code generation tools; we encourage broader use of these tools in protocol implementations, as incorrect input handling jeopardizes other properties of an implementation.

III. Background Formalisms

A. Computability Bounds and the Chomsky Hierarchy

Noam Chomsky classified formal grammars in a containment hierarchy according to their expressive power, which correlates with the complexity of the automaton that accepts exactly the language a grammar generates, as shown in Fig. 1 [19].

Within this hierarchy, one class of automaton can decide an equivalently powerful language or a less powerful one, but a weaker automaton cannot decide a stronger language. For example, a pushdown automaton can decide a regular language, but a finite state machine cannot decide a context-free language. Thus, formal input validation requires an automaton (hereafter, parser) at least as strong as the input language. It is a useful conceit to think of a protocol grammar in terms of its place in the Chomsky hierarchy, and the processor and code

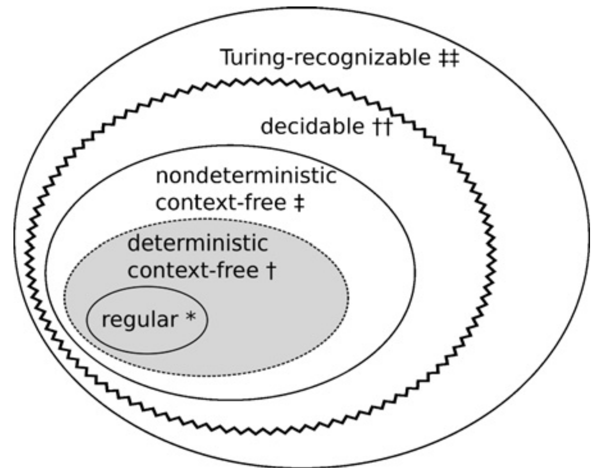

Fig. 1. Chomsky hierarchy of languages according to their expressive power. Languages correspond to grammars and automata as follows.
⋆ regular grammars, regular expressions, finite state machines;
† unambiguous CFGs, deterministic pushdown automata;
‡ ambiguous CFGs, nondeterministic pushdown automata;
†† context-sensitive grammars/languages, linear bounded automata;
‡‡ recursively enumerable languages, unrestricted grammars, Turing machines;
The shaded area denotes classes for which the equivalence problem is DECIDABLE.

that accept input in terms of machine strength, while being conscious of their equivalence.

Recursively enumerable languages are undecidable, which presents a serious implementation problem: the Turing machine that accepts a given recursively enumerable language, or recognizer[3] language (which is stronger than context-sensitive but weaker than recursively enumerable), rejects strings not in the language, and is guaranteed to halt for that language, halts in an accepting state on all strings in the language, but either rejects or fails to halt on inputs not in the language. All weaker language classes are decidable; their equivalent automata always terminate in an accept or reject state [10], [19]. A recursively enumerable protocol language is thus a security risk, since malicious input could cause its parser to fail to halt—a syntactic denial of service—or perform arbitrary computation. The rubric we derive from these bounds on expressiveness—use a sufficiently strong parser for an input language, but no stronger—is echoed in the W3C's Rule of Least Power: Use the least powerful language suitable for expressing information, constraints, or programs on the World Wide Web [20].

Parsers also exhibit certain safety and liveness properties (after Lamport [4]). Soundness is a safety property; a sound parser only accepts strings in its corresponding language, and rejects everything else. Completeness is a liveness property; a complete parser accepts every string in its corresponding language. Termination is also a safety property; a terminating parser eventually halts on every string presented to it, whether that string belongs to its language or not.

Two other decidability problems influence our analysis: the context-free equivalence and containment problems. Given

[2]Albeit with notable exceptions, such as XSLT [11], [12], HTML5+CSS3 (shown to be undecidable by virtue of its ability to implement Rule 110 [13], [14]), and PDF (for many reasons, including its ability to embed Javascript [15]).

[3]Compare with the decider, a turing machine that accepts strings in a recursive.

two arbitrary CFGs, G and H, both $\mathcal{L}(G) = \mathcal{L}(H)$ and $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ are undecidable [10], except for a particular construction detailed in [21]. The CFGs form two disjoint sets: deterministic and nondeterministic, corresponding to deterministic and nondeterministic pushdown automata respectively. All unambiguous CFGs are deterministic [22], and the equivalence problem for deterministic CFGs is decidable (though the containment problem is not) [9].

Any grammar in which the value of an element in the string influences the structure of another part of the string is at least context-sensitive [23]. This applies to most network protocols and many file formats, where length fields, e.g., the Content-Length field of an HTTP header [24] or the IHL and Length fields of an IPv4 datagram [25], are commonplace. Programming language grammars that support statements of the form *if B1 then if B2 then S1 else S2*, e.g., Javascript [26], are nondeterministic context-free (at best) due to the ambiguity the dangling else problem introduces [27]; if the shift-reduce conflict is resolved without adding braces or alternate syntax (e.g., elif or end if), the resulting grammar is noncontext-free. Conveniently, the PostgreSQL, SQLite, and MySQL database engines all use LR grammars, which are deterministic context-free [28]. Membership tests for certain subclasses of LR (e.g., LALR, LR(k), etc.) and approximate ambiguity detection methods exist [29]; however, determining whether an arbitrary CFG is unambiguous is undecidable [30].

B. Modeling Communication From a Security Standpoint

Shannon [31] proposed a block-diagram model to describe systems which generate information at one point and reproduce it elsewhere. In it, an information source generates messages (sequences drawn from an alphabet); a transmitter encodes each message into a signal in an appropriate form for the channel over which it can pass data, then sends it; a receiver decodes signals into reconstructed messages; and a destination associated with the receiver interprets the message. The engineering problem of maximizing encoding efficiency motivated Shannon's work; he regarded the meanings of messages as outside the scope of this transmission model. Nevertheless, social scientists such as Schramm [32] and Berlo [33] expanded the transmission model to incorporate semantic aspects of communication. Schramm recast the destination as an interpreter, which takes actions according to the decoded message's semantic content, and replaced Shannon's one-way message path with a bidirectional one; Berlo emphasized the difficulty of converting thoughts into words and back, particularly when sender and receiver differ in communication ability. These insights, combined with Hoare's axiomatic technique for defining programming language semantics [34], have surprising implications for the practice of computer security.

When a destination extracts a different meaning from a decoded message than the one the source intended to transmit, the actions the destination performs are likely to diverge—perhaps significantly—from what the source expected. In human communication, it is difficult to evaluate whether an unexpected response signifies a failure in transmission of meaning or that the source's assessment of what behavior to expect from the destination was wrong. In computer science, however, we can make formal assertions about the properties of destinations (i.e., programs), reason about these properties, and demonstrate that a program is correct up to decidability [34]. When a destination program's semantics and implementation are provably correct, whether or not it carries out its intended function [34] is a question of whether the destination received the intended message. If a verified destination's response to a source's message M does not comport with the response that deduction about the program and M predict, the receiver has decoded something other than the M that the transmitter encoded. In practice, this situation is all too frequent between different implementations of a protocol.

Berlo's and Schramm's adaptations rightly drew criticism for their focus on encoding and decoding, which implied the existence of some metric for equivalence between one person's decoder and the inverse of another person's encoder. However, in transmissions over computer networks, where both source and destination are universal Turing machines, we can test the equivalence of these automata if they are weak enough; if they are nondeterministic context-free or stronger, their equivalence is undecidable. Points of encoder–decoder inequivalence—specifically, instances where, for a message M, an encoding function $\mathcal{E}$, and a decoding function $\mathcal{D}$, $\mathcal{D}(\mathcal{E}(M)) \neq M$—can cause the destination to take some action that the source did not anticipate. An attacker who can generate a signal $\mathcal{E}(M)$ such that $\mathcal{D}(\mathcal{E}(M)) \neq M$ can take advantage of this inequivalence. Indeed, many classic exploits, such as buffer overflows, involve crafting some $\mathcal{E}(M)$—where the meaning of M, if any, is irrelevant [4]—such that applying $\mathcal{D}$ to $\mathcal{E}(M)$, or passing $\mathcal{D}(\mathcal{E}(M))$ as an input to the destination, or both, elicits a sequence of computations advantageous to the attacker (e.g., opening a remote shell).

Naturally, an attacker who can alter $\mathcal{E}(M)$ in the channel, or who can modify M before its encoding, can also elicit unexpected computation. The former is a man-in-the-middle attack; the latter is an injection attack. Both affect systems where the set of messages that the source can generate is a subset of those on which the destination can operate.

Note that we do not consider situations where $\mathcal{D}(\mathcal{E}(M)) = M$ but different destinations respond to M with different actions; these constitute divergent program semantics, which is relevant to correctness reasoning in general but outside the scope of this paper. We are only interested in the semantics of $\mathcal{D}$ and $\mathcal{E}$.

IV. Exploits As Unexpected Computation

Sending a protocol message is a request for the receiving computer to perform computation over untrusted input. The receiving computer executes the decoding (parsing) algorithm $\mathcal{D}(M)$, followed by (i.e., composed with) subsequent operations $\mathcal{C}$ conditional on the result of $\mathcal{D}$; thus, $\mathcal{E}(M) \rightarrow \mathcal{D}(\mathcal{E}(M)) \cdot \mathcal{C}(\mathcal{D}(\mathcal{E}(M)))$. It is never the case that simply parsing an input from an untrusted source should result in malicious code execution or unauthorized disclosure of sensitive

[4]This phenomenon suggests that Grice's maxim of relation [35], be relevant, applies to the pragmatics of artificial languages and natural ones.

information; yet, this is the basis of most effective attacks on modern networked computer systems, specifically because they permit, though they do not expect, the execution of malicious algorithms when provided the corresponding input. That this computation is unexpected is what leads to such vulnerabilities being considered exploits, but ultimately, the problem constitutes a failure in design. Whether implicitly or explicitly, designers go to work with a contract [36] in mind for the behavior of their software, but if the code does not establish and enforce preconditions to describe valid input, many types of exploits are possible.

This behavior is especially harmful across layers of abstraction and their corresponding interfaces, since in practice these layer boundaries become boundaries of programmers' competence.

A. Injection Attacks

Injection attacks target applications at points where one system component acquires input from a user in order to construct an input for another component, such as a database, a scripting engine, or the DOM environment in a browser. The attacker crafts an input to the first component that results in the constructed input producing some computation in the second component that falls outside the scope of the operations the system designer intended the second component to perform. Some examples:

Example 1 (Command Injection): Functions such as system() in PHP, Perl, and C; nearly all SQL query execution functions; and Javascript's eval() take as argument a string representation of a command to be evaluated in some execution environment (here, the system shell, a database engine, and the Javascript interpreter respectively). Most such environments support arbitrary computation in their own right, though developers only intend their systems to use a very limited subset of this functionality. However, when these functions invoke commands constructed from unvalidated user input, an attacker can design an input that appends additional, unauthorized commands to those intended by the developer—which the environment dutifully executes, using the same privileges afforded to the desired commands [37].

Example 2 (HTTP Parameter Pollution): RFC 3986 [38] observes that the query component of a URI often contains key=value pairs that the receiving server must handle, but specifies nothing about the syntax or semantics of such pairs. The W3C's form-urlencoded media type [39] has become the de facto parameter encoding for both HTTP GET query strings and HTTP POST message bodies, but parameter handling semantics are left to implementer discretion. Idiosyncratic precedence behavior for duplicate keys, in a query string or across input channels, can enable an attacker to override user-supplied data, control web application behavior, and even bypass filters against other attacks [40].

All types of injection leverage a weak boundary between control and data channels [41] to modify the structure, and thereby the execution semantics, of an input to an application component [21], [41]. Halfond *et al.* [42] enumerate many heuristic injection defenses; in Section V-A we describe parse tree validation, a verifiable defense technique. There are several categories of defense against injection: escaping, which attempts to transform user input that might alter the structure of a subsequently constructed input into a string-literal equivalent; tainting, which flags user input as untrusted and warns if that input is used unsafely; blacklisting of known malicious inputs; and programmatic abstraction, which provides control channel access through an API and relegates user input to the data channel [42]. Another technique, parse tree validation, passes constructed inputs through a validator that parses them, compares the resulting parse tree to a set of acceptable candidate parse trees, and rejects inputs whose structure is not in that set.

B. Other Attack Surfaces

Other attack vectors blur the boundaries between control and data channels in subtler ways; rather than targeting the higher level languages that injection exploits, they take advantage of input handling failure modes to alter the machine code or bytecode in an already-executing process. Many such attacks, e.g., shellcode attacks [43], contain a sequence of opcodes that are written to a location within the process's address space and executed by means of a jump from an overwritten stack frame return address; other techniques, such as return-to-libc [44] and its generalization, return-oriented programming [45], [46], overwrite the return address to point to a function or a code fragment (a.k.a. gadget, e.g., in the program's code section, or in a library such as libc) not meant to be a part of the stack-backed control flow and adjacent memory to contain any arguments the attacker wants to pass to that function, enabling arbitrary code execution even on platforms with nonexecutable stacks [47].

Example 3 (Buffer Overflows): When a function designed to write data to a bounded region of memory (a buffer) attempts to write more data than the buffer can contain, it may overwrite the values of data in adjacent memory locations—possibly including the stack frame return address [48] or a memory allocator's heap control structures [49]–[51]. Constraining such a function's input language to values that the function cannot transform into data larger than the buffer can prevent an overflow, although the presence of format string arguments (see below) can complicate matters.

Example 4 (Format String Attacks): Certain C conversion functions permit placeholders in their format string argument which interpolate subsequent arguments into the string the function constructs. If a process allows an attacker to populate the format string argument, he can include placeholders that let him inspect stack variables and write arbitrary values to arbitrary memory locations [52]. Other languages that support format strings exhibit similar vulnerabilities [53], and languages implemented in C, such as PHP, can succumb indirectly if unsafe input reaches a format string argument in the underlying implementation [54]. Fortunately, C's placeholder syntax is regular, and since the regular languages are closed under complement [10], it is easy to define a positive validation routine [42] which admits only user input that contains no formatting placeholders.

Thus, we see that hardening input routines, so that they do not provide subsequent operations with arguments that violate

those operations' preconditions or fail in ways that permit an attacker to execute arbitrary code, is at the core of all defensive coding practices. We now examine in detail the mechanics of validating input languages of various classes in a provable and tractable fashion.

V. Provably Correct Input Validation

Despite the majority of work in this area focusing on injection attacks, formal language theoretic input validation offers security protections against a much wider range of exploits. Any attack that exploits a process's parsing such that it accepts an input that does not conform to the valid grammar of the intended protocol can and should be prevented via strict validation of inputs.

A. Injection Attacks and Context-Free Parse Tree Validation

Dejector [21] presented a context-free parse tree validation approach to preventing SQLIA.[5] It introduced a formal construction for restricted sublanguages of SQL;[6] using this approach, validating an SQL query consists of testing it for membership in the sublanguage. Given a set of known-good queries—derived, for instance, from the string-interpolated query templates that an application programmer has defined for a particular application—and the formal grammar for the appropriate dialect of SQL, Dejector transforms the SQL grammar into a subgrammar that contains only the rules required to produce exactly the queries in the known-good set.[7] Strings recognized by the subgrammar are guaranteed to be structurally identical to those in the known-good set—a validity metric attested throughout the injection attack literature [57], [58]. The subgrammar is then used with a parser generator such as bison or ANTLR to produce a recognizer for the sublanguage. Notably, this automaton is exact rather than heuristic (as in [55]) or approximate (as in [59] and [60]), and has the optimizing effect of comparing inbound queries to all known-good structures simultaneously.

Subsequent research has produced improvements to the original approach, primarily focused on identifying the initial legitimate-query set and automatically integrating validation into an application. Unfortunately, each of these efforts suffers from flaws which prevent them from guaranteeing correct validation or correct application behavior. These include:

[5]The technique was independently discovered by a number of research teams. References [55] and [56] published similar approaches around the same time; [57] popularized the idea, but Dejector has thus far been mostly overlooked by the academic community due to its publication at a hacker conference. This is, to our knowledge, the first peer-reviewed description of Dejector, with an emphasis placed on the features that distinguish it from later attempts to implement these core ideas.

[6]Su and Wassermann [58] independently arrived at the same construction.

[7]Unused production rules are removed, as are unused alternatives from retained production rules. If the rule $A ::= B|C$ appears in the grammar, but the known-good set only requires the application of the $A \Rightarrow C$ branch, the corresponding subgrammar rule is $A ::= C$. Note that for grammars where a nonterminal whose right-hand side contains more than one alternative appears on the right-hand side of more than one nonterminal that appears in the parses of known-good queries, these alternatives must be distinguished in the subgrammar.

1) *Insufficiently Strong Automaton:* Several automata-based validators [59]–[63] model the set of acceptable queries using a finite state machine, following the approach of Christensen *et al.* [64], wherein static analysis of calls to methods that issue SQL queries yields a flow graph representing possible generated strings, which is then widened to a regular language for tractability. Sun and Besnozov identify cases where such FSA models generate false-positive reports [65], and indeed Wassermann *et al.* concede that their approximation of the set of legitimate query strings is overly permissive. However, they assert:

> In practice, we do not find a function that concatenates some string, the return value of a recursive call to itself, and another string (which would construct a language such as $\{({}^n a)^n\}$), so this widening step does not hurt the precision of the analysis.

We examined the bison grammar that generates the PostgreSQL parser and, regrettably, discovered four such functions. The right-hand sides of the production rules select_with_parens and joined_table contain the precise parenthesis-balancing syntax that Wassermann *et al.* claimed not to find in practice. Unbalanced parentheses alone are sufficient to trigger those vulnerabilities classified in the taxonomy of Halfond *et al.* as illegal/logically incorrect queries [42].

The other functions we found are subtler and more troubling. The right-hand side of the production common_table_expr, which can precede SELECT, INSERT, UPDATE, or DELETE statements, contains the sequence '(' PreparableStmt ')'; a PreparableStmt is itself a SELECT, INSERT, UPDATE or DELETE statement. Furthermore, the a_expr and c_expr productions, which recognize unary, binary, and other expressions—such as x NOT $NULL$, x $LIKE$ y, and all arithmetic expressions—are mutually recursive. These productions appear throughout the PostgreSQL grammar, and are the grammatical targets of nearly every category of SQLIA, since user-supplied inputs typically correspond to productions on the right-hand side of an a_expr.

Thus, while tools using this methodology have performed well against SQLIA suites such as the AMNESIA testbed [66], [67], we question their efficacy against attacks that deliberately target the impedance mismatch between a generated FSA model and an underlying SQL grammar.

2) *Validator and Database Use Different Grammars:* Many parse tree validation approaches properly represent the set of acceptable structures using a CFG, but derive their acceptable-structure set from a grammar other than that of the target database system, possibly introducing an impedance mismatch. SQLGuard [56] compares parse trees of queries assembled with and without user input, using the ZQL parser [68]; CANDID [57] uses a standard SQL parser based on SQL ANSI 92 standard, augmented with MySQL-specific language extensions; SQLPrevent [65] uses ANSI SQL but does not mention which version. Others only state that the grammars they use are context-free [58], [69], [70].

While it is possible to demonstrate the equivalence of two LR grammars, none of these authors have provided equivalence proofs for their implementation grammars and the SQL

dialects they aim to validate. Dejector sidesteps this problem by directly using PostgreSQL's lexer and bison grammar. Dejector's drawback is that its implementation is coupled not only to the database distribution, but the specific parser revision; however, it prevents an attacker from constructing an input that looks right to the validator but yields unwanted behavior when it reaches the database. As an example, CVE-2006-2313 and CVE-2006-2314 describe a relevant vulnerability in PostgreSQL multibyte encodings [71], [72]. An attacker could craft a string that an encoding-unaware validator (i.e., one that assumes input to be in ASCII, Latin-1 or some other single-byte encoding) accepts, but which a server using a multibyte encoding (UTF-8, Shift-JIS, etc.) parses in such a way as to terminate a string literal early. We examine such parse tree differential attacks in more detail in Section VI.

B. Parse Tree Validation in the Context-Sensitive Languages

Bleichenbacher [6] presented an RSA signature forgery attack against PKCS#1 implementations that do not correctly validate padding bytes. We show that PKCS#1 is context-sensitive and can be validated in the same fashion as SQL, using an attribute grammar representation [73].

Theorem 1: PKCS#1 is context-sensitive.

Lemma 1: Upper bound: a linear-bounded automaton for PKCS#1.

Proof: Let $P = \{w^n | w$ is a hexadecimal octet, n is the size of the RSA modulus in bits, and w^n = '00' '01' 'FF'$^{n-len(hash)-len(d.e.)-3}$ '00' digest-encoding hash, where digest-encoding is a fixed string $\in \{0, 1\}$ as specified in RFC 2313 [74] and hash is a message hash $\in \{0, 1\}$ of length appropriate for the digest-encoding$\}$. We define a linear-bounded automaton, A_P, that accepts only strings in P. The length of A_P's tape is n, and it has states $q_0, q_1, ...q_{67}$ and a reject state, q_R. q_{67} is the start state.

1) Go to the leftmost cell on the tape.
2) Consume octet 00 and transition to state q_{66}. If any other octet is present, transition to q_R and halt.
3) Consume octet 01 and transition to state q_{65}. If any other octet is present, transition to q_R and halt.
4) Consume FF octets until any other octet is observed, and transition to state q_{64}. (If the first octet following the 01 is anything other than FF, transition to q_R and halt.)
5) Simulate regular expression matching of the fixed digest-encoding strings (as described in the attribute grammar in the next subsection) over the next 15-19 octets as follows.
   a) MD2 sequence $\rightarrow q_{15}$.
   b) MD5 sequence $\rightarrow q_{15}$.
   c) SHA-1 sequence $\rightarrow q_{19}$.
   d) SHA-256 sequence $\rightarrow q_{31}$.
   e) SHA-384 sequence $\rightarrow q_{47}$.
   f) SHA-512 sequence $\rightarrow q_{63}$.
   g) No match $\rightarrow q_R$.
6) Until q_0 is reached, or the rightmost end of the tape is reached, apply the following procedure.
   a) Consume an octet.
   b) $q_n \rightarrow q_{n-1}$.
7) If in state q_0 and the tape head is at the rightmost end of the tape, accept. Otherwise, reject.

Because P can be described by a linear-bounded automaton, it is therefore at most context sensitive. ■

Lemma 2: Lower bound: PKCS#1 is not context-free.

Proof: We show that P is noncontext-free using the context-free pumping lemma, which states that if L is a context-free language, any string $s \in L$ of at least the pumping length p can be divided into substrings $vwxyz$ such that $|wy| > 0$, $|wxy| \leq p$, and for any $i \geq 0$, $vw^ixy^iz \in A$ [10].

As stated above, n is the size of the RSA modulus in bits. (n can vary from case to case; different users will have different-sized RSA moduli, but the grammar is the same no matter the size of n.) Neither w nor y can be any of the fixed bits, 00, 01 and 00, since the resulting string would be too long to be in P. Nor can w or y correspond to any part of the hash, as the pumping lemma requires that w and y can be pumped an arbitrary number of times, and eventually the length of the hash alone would exceed n. Indeed, since n is fixed, the only way to pump s without obtaining a string that is either too long or too short would be if both w and y were the empty string. However, the pumping lemma requires that $|wy| \geq 0$, and thus P cannot be context-free. Since P is at most context-sensitive and must be stronger than context-free, P is therefore context-sensitive. ■

1) *An Attribute Grammar for PKCS#1:* Attribute grammars and parsing expression grammars [75] are concrete formalisms that are commonly used to implement parsers and generators for context-sensitive languages. Parsing expression grammars can describe some context-sensitive languages, e.g., the well-known $\{a^nb^nc^n : n \geq 1\}$, but not all of them; attribute grammars are sufficient to describe any context-sensitive language. As Finney pointed out, the Bleichenbacher padding attack produces strings which vulnerable implementations interpreted as valid encodings, but which are not actually members of P because they have too few padding bytes and extra data beyond the message hash. Therefore, a sound and complete attribute-grammar-based parser for PKCS#1 functions as a validator for PKCS#1 and defeats the Bleichenbacher attack by rejecting its crafted strings. Since the follow-on attack of Izu *et al.* also assumes the same implementation error as in the original attack, our validator also defeats their mathematically more sophisticated attack.

Note that here we describe a validator for PKCS#1 *in toto*, rather than a restricted sublanguage of P. One could define a restricted sublanguage of P—for instance, one that does not support the deprecated MD5 and MD2 hash functions—by pruning out the corresponding production rules and any alternatives that contain references to them, but here we focus on the language in its entirety.

The following attribute grammar, where ⟨T⟩ represents any valid octet from 00 to FF, generates strings in P for arbitrary n:

```
⟨S⟩ ::= 00 01 ⟨FFs⟩ 00 ⟨ASN.1⟩
Valid(⟨S⟩) ← Valid(⟨ASN.1⟩) & Len(⟨FFs⟩) = n − Len(⟨ASN.1⟩) − 3
⟨FFs⟩ ::= FF FF FF FF FF FF FF FF
Len(⟨FFs⟩) ← 8
              | ⟨FFs⟩2 FF
Len(⟨FFs⟩) ← (Len(⟨FFs⟩2) + 1)
```

⟨ASN.1⟩ ::= ⟨Digest-Algo⟩ ⟨Hash⟩
Valid(⟨ASN.1⟩) ← (HashLen(⟨Digest-Algo⟩) = Len(⟨Hash⟩))
Len(⟨ASN.1⟩) ← (Len(⟨Digest-Algo⟩) + Len(⟨Hash⟩))
⟨Digest-Algo⟩ ::= ⟨MD2⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨MD2⟩)
Len(⟨Digest-Algo⟩) ← 18
| ⟨MD5⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨MD5⟩)
Len(⟨Digest-Algo⟩) ← 18
| ⟨SHA-1⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨SHA-1⟩)
Len(⟨Digest-Algo⟩) ← 15
| ⟨SHA-256⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨SHA-256⟩)
Len(⟨Digest-Algo⟩) ← 19
| ⟨SHA-384⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨SHA-384⟩)
Len(⟨Digest-Algo⟩) ← 19
| ⟨SHA-512⟩
HashLen(⟨Digest-Algo⟩) ← HashLen(⟨SHA-512⟩)
Len(⟨Digest-Algo⟩) ← 19
⟨MD2⟩ ::= 30 20 30 0C 06 08 2A 86 48 86 F7 0D 02 02 05 00 04 10
HashLen(⟨MD2⟩) ← 16
⟨MD5⟩ ::= 30 20 30 0C 06 08 2A 86 48 86 F7 0D 02 05 05 00 04 10
HashLen(⟨MD5⟩) ← 16
⟨SHA-1⟩ ::= 30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14
HashLen(⟨SHA-1⟩) ← 20
⟨SHA-256⟩ ::= 30 31 30 0D 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
HashLen(⟨SHA-256⟩) ← 32
⟨SHA-384⟩ ::= 30 41 30 0D 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30
HashLen(⟨SHA-384⟩) ← 48
⟨SHA-512⟩ ::= 30 51 30 0D 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40
HashLen(⟨SHA-512⟩) ← 64
⟨Hash⟩ ::= $\langle \mathrm{T} \rangle^{16}$
Len(⟨Hash⟩) ← 16
| $\langle \mathrm{Hash} \rangle_2$ ⟨T⟩
Len(⟨Hash⟩) ← $Len(\langle Hash \rangle_2) + 1$

VI. Parse Tree Differential Analysis

We observe that, while different implementations of the same specification should process input and perform tasks in effectively the same way as each other, it is often the case that different implementations parse inputs to the program (or messages passed internally) differently depending on how the specification was interpreted or implemented. Such implementations provide distinct dialects of a protocol. While these dialects may be mutually intelligible for the purpose of nonmalicious information exchange, prior security assumptions may fail.

We have developed a powerful technique to enhance code auditing and protocol analysis, known as the parse tree differential attack [8], wherein we give two different implementations of the same specification identical state and input parameters, consider their decodings as concrete parse trees, and enumerate the differences between the trees. Deviations between the trees indicate potential problems, e.g., an area of implementor discretion due to specification ambiguity or an implementation mistake.

Looking back to the work of Shannon *et al.* in Section III-B, the goal of a parse tree differential attack is to find combinations of M_{source}, $\mathcal{E}_{\text{source}}$, and $\mathcal{D}_{\text{destination}}$ such that $M \neq \mathcal{D}(\mathcal{E}(M))$, with M semantically valid for the source and $\mathcal{D}(\mathcal{E}(M))$ semantically valid for the destination, where the destination's response to $\mathcal{D}(\mathcal{E}(M))$ includes computations that its response to M would not have. The set

$$\{M_A \cup M_B \mid M_A \neq \mathcal{D}_B(\mathcal{E}_A(M_A)), M_B \neq \mathcal{D}_A(\mathcal{E}_B(M_B))\}$$

describes a lower bound on the set of vulnerabilities present on the attack surface of the composed system that has implementations (i.e., processes) A and B as endpoints of a common channel (after Howard *et al.* [76]).

A. Attack Surface Discovery

We have used edge cases identified by parse tree differential analysis to isolate serious vulnerabilities in X.509 [77]. We found many instances where two implementations of the X.509 system behaved differently when given the same input, in such a way that these differences led to a certificate authority signing a certificate that it viewed as being granted one privilege, while the client-side application (the web browser) parsed the same input in a manner yielding different security assertions, leading to a compromise of the system [8].

Example 5 (Null Terminator Attack): The attacker presents a certificate signing request to a certificate authority (CA) that will read the Common Name as www.paypal.com\x00.badguy.com and return a signed certificate for this Subject CN. The message that this certificate represents is M, and the certificate itself is $\mathcal{E}(M)$. Now present $\mathcal{E}(M)$) to a browser vulnerable to the null terminator attack. Although the CN field's value in M is www.paypal.com\x00.badguy.com, its value in $\mathcal{D}(\mathcal{E}(M))$ is www.paypal.com.

In this case, the decoder at the CA correctly interprets the CN as a Pascal-style string (which can include the \x00 character), compares its reading of the CN with the credentials presented by the source, and responds with an encoding of a message incorporating this valid-but-peculiar CN. Little does the destination know, other destinations' decoders interpret the CN as a C-style string, for which \x00 is an end-of-string indicator, and decode the CA's encoding into a signed message vouching that the certificate is valid for www.paypal.com!

B. Other Applications of Parse Tree Differentials

In certain settings, aspects of protocol implementation divergence are of particular sensitivity; a prime example is anonymity systems. Prior work has shown that the anonymity provided by a lower layer tool can be compromised if higher layer differences are revealed to an attacker; the EFF's Panopticlick tool demonstrates how to use web browser identifiers to whittle away the assurances offered by low-level anonymity systems such as Tor [78]. The potential for an attacker to perform parse tree differential analysis of common client application implementations of standard protocols a priori allows her to generate a codebook of sorts, consisting of the inputs which, when sent to the unsuspecting user, will trigger the user's client (web browser, etc.) to respond in a way that will enable the attacker to partition the anonymity set [79]. Similarly, the use of a parse tree differential analysis tool may enhance other fingerprinting-based attacks.

A more indirect means of using parse tree differentials as oracles appears in Clayton's work on British Telecom's CleanFeed anti-pornography system [80]. He constructed TCP packets with a specially chosen TTL value which, if actually used, would leverage the CleanFeed proxy system's traffic-redirection behavior against the behavior of noninterdicted traffic so as to selectively reveal exactly which IP addresses hosted material that BT was attempting to block!

Notably, Clayton's attack makes use of three separate protocols—TCP, IP, and ICMP—being used by multiple systems (a user's, BTs, and that of a banned site). This highlights the empirically well-known observation that composed systems tend to have characteristic behaviors that result from composition and are not obviously inherent in the individual components. In critical applications (such as an anonymity system used to evade violent repression), such behaviors can be deadly. To quote a hacker maxim, Composition Kills.

1) *A Well-Defined Order on Parse Tree Differentials:* Consider a parse tree differential attack executed between two different implementations of the same protocol a zeroth-order parse tree differential. It has two steps, protocol encoding and protocol decoding.

Now consider a parse tree differential attack executed between two different implementations of two different protocols, e.g., ASN.1 → HTTP. (e.g., X generates ASN.1 which is transformed into HTTP which is parsed by Y). The transformation between one protocol and another is a point of interest; can, for instance, malformed ASN.1 be generated with respect to the transformation function to HTTP such that Y performs some unexpected computation? This is a first-order parse tree differential. It has three steps: protocol encoding, protocol transformation (to protocol') and protocol' decoding.

The construction extends recursively.

VII. Why Johnny Cannot Detect

One arguably nonprincipled but practically common form of composition is that of adding an intrusion detection/prevention system (IDS) to a target known to be susceptible to exploitation. The IDS monitors the target's inputs and/or state, models the target's computation, and is expected to catch the exploits. This design obviously relies on the ability of the IDS to match at least those aspects of the target's input processing that serve as attack vectors; without such matching the IDS does not reduce insecurity, and may in fact increase it by adding exploitable bugs of its own. Far from being merely theoretical, the latter is a hard reality well-known to security practitioners on both the attack and defense sides (see [81]).

The language-theoretic and computational magnitude of the challenge involved in constructing such an effective matching in this de facto composed design should by now be clear to the reader, as it requires approaching de facto computational equivalence between the IDS and the target input handling units. The first work [82] to comprehensively demonstrate the fundamental weakness of network intrusion detection systems (NIDS) was, predictably, based on hacker intuitions. These intuitions were likely informed by previous use of TCP/IP stack implementation differences for system fingerprinting in tools like Nmap, Xprobe, and Hping2 (e.g., [83] methodically explores the differences in OS network stacks' response to various ICMP features). Subsequent research established that the only hope of addressing this weakness was precise matching of each target's session (re)assembly logic by the NIDS (e.g., [84]–[86]).

In host-based intrusion detection systems, the problem of matching the defending computation with the targeted computation is no less pronounced. For example, Garfinkel [87] enumerates a number of traps and pitfalls of implementing a system call-monitoring reference monitor and warns that duplicating OS functionality/code should be avoided at all costs. We note that isolating the reference monitor logic from the rest of the system would seem advantageous were it possible to validate the matching between the system's own computation and the isolated, duplicated computation; however, as we have seen, such validation could easily be undecidable.

In a word, hardening a weak system by composing it with a monitor that replicates the computation known or suspected to be vulnerable likely attempts to convert an input-validation kind of undecidable problem into a computational-equivalence undecidable problem—hardly an improvement in the long run, even though initially it might appear to gain some ground against well-known exploits. However, it leaves intact the core cause of the target's insecurity, and should not therefore be considered a viable solution.

One common approach for modeling program behavior involves sequences of system calls [88], [89]. Because system calls represent the method by which processes affect the external world, these sequences are thought to provide the most tangible notion of system behavior. Despite their apparent success in detecting anomalies due to attacks, such models have several shortcomings, including susceptibility to mimicry attacks [90]; an attacker can keep the system within some epsilon of the normal patterns while executing calls of their choosing. This problem suggests that we should investigate the extraction and use of a more fine-grained notion of program activity. Note that our goal is not to criticize system call approaches for being susceptible to mimicry attacks; instead, the lesson we should learn is that relatively large amounts of work can happen between system calls, and it is the more precise nature of this activity that can help inform models of program behavior.

Popular flavors of model or anomaly based intrusion detection often offer only very slight deltas from each other; Taylor and Gates [91] supply a good critique of current approaches, and a recent paper by Sommer and Paxson also explores the reasons why we as a community might not successfully use machine learning for intrusion detection [92]. The prevailing approach to detection (matching sequences of system calls) is a glorified form of the oft-critized regular expression string matching used in misuse signature-based systems like Snort and Bro.

An impressive number of RAID, CCS, and Oakland papers have spilled a lot of digital ink offering slight twists or improvements on the original system call sequence model proposed by Denning and matured by Forrest, Somayaji

et al. [88], [93], [94]. This follow-on pack of work considers, in turn, changes that include: longer sequences, sequences with more context information (e.g., instruction pointer at time of call, arguments, machine CPU register state, sets of open files and resources), anomalous system call arguments, cross-validation of system call sequences across operating systems, and other various insignificant changes in what information is examined as the basis of a model.

The most natural next step was to attempt to characterize normal behavior, abnormal behavior, and malware behavior using control-flow graph structures. From this perspective, sequences of system calls are very simple graphs with a linear relationship.

Unfortunately, this move toward more complicated models of representing execution behavior reveal just how limited we are in our expected success. When viewed from the pattern of language-theoretic equivalence, this style of intrusion detection is essentially a problem of matching grammars, and it suffers from the same limitations as proving that two protocol implementations of sufficient complexity actually accept the same strings.

The intrusion detection community overlooks this critically important point in its search for ever more efficient or representative models of malicious (or benign) behavior. Adding incremental adornments to a language model will not result in a dramatic advancement of our ability to detect malicious computation; it can only serve to increase the complexity of the language—and hence increase the difficulty of showing that the particular model accepts some precise notion of malicious or abnormal. This is a counter-intuitive result: initiatives aimed at improving the power of an IDS model actually detract from its ability to reliably recognize equivalent behavior. In this case, more powerful maps to less reliable.

We note that, to the best of our knowledge, Schneider [95] comes closest to considering the limits of security policy enforceability as a computation-theoretic and formal language-theoretic phenomenon, by matching desired policy goals such as bounded memory or real-time availability to classes of automata capable of guaranteeing the acceptance or rejection of the respective strings of events. In particular, Büchi automata are introduced as a class of security automata that can terminate insecure executions defined by the Lamport's safety property: execution traces excluded from the policy can be characterized as having a (finite) set of bad prefixes (i.e., no execution with a bad trace prefix is deemed to be safe).

Schneider's approach connects enforceable security policies with the language-theoretic properties of the system's language of event traces. We note that the next step is to consider this language is an input language to the automaton implementing the policy mechanism, and to frame its enforcement capabilities as a language recognition problem for such trace languages.

VIII. Future Work

Our future work will integrate existing work on generation of verifiable, guaranteed-terminating parsers [16]–[18] with verification of finite-state concurrent systems and the work of Bhargavan *et al.* [96] on the use of refinement types to carry security invariants (and, by extension, input-language preconditions) in order to develop a complete verified network stack that is compositionally correct from end to end. We also plan to build on previous work in automated implementation checking, such as aspier [97], to develop automated parse tree differential analysis tools (akin to smart fuzzers) for the benefit of security auditors.

Acknowledgment

The authors would like to thank E. Feustel for his observations on the security of composed systems, D. Kaminsky for his collaboration in the analysis of flaws in ASN.1 parsers, A. Bogk, R. Farrow, D. McCardle, J. Oakley, F. Piessens, A. Shubina, and S. W. Smith for their helpful suggestions on earlier versions of this paper, and N. Borisov, N. Kisserli, F. Lindner, D. McIlroy, and D. Molnar for their insightful conversations during the process of this research.

References

[1] D. Geer, "Vulnerable compliance," *login: The USENIX Mag.*, vol. 35, no. 6, Dec. 2010.
[2] F. "FX" Lindner, "The compromised observer effect," *McAfee Security J.*, vol. 6, 2010.
[3] D. J. Bernstein, "Some thoughts on security after ten years of qmail 1.0," in *Proc. ACM CSAW*, 2007, pp. 1–10.
[4] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977.
[5] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.
[6] H. Finney, "Bleichenbacher's RSA signature forgery based on implementation error," Aug. 2006.
[7] T. Izu, T. Shimoyama, and M. Takenaka, "Extending Bleichenbacher's forgery attack," *J. Inform. Process.*, vol. 16, pp. 122–129, Sep. 2008.
[8] D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI layer cake: New collision attacks against the global X.509 infrastructure," in *Financial Cryptography*. Berlin, Germany: Springer, 2010, pp. 289–303.
[9] G. Sénizergues, "L(A) = L(B)? Decidability results from complete formal systems," *Theor. Comput. Sci.*, vol. 251, nos. 1–2, pp. 1–166, 2001.
[10] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed., International ed. Clifton Park, NY: Thompson Course Technology, 2006.
[11] S. Kepser, "A simple proof for the turing-completeness of XSLT and xQuery," in *Proc. Extreme Markup Lang.*, 2004.
[12] R. Onder and Z. Bayram, "XSLT version 2.0 is turing-complete: A purely transformation based proof," in *Proc. Implementation Appl. Automata*, LNCS 4094. 2006, pp. 275–276.
[13] E. Fox-Epstein. (2011, Mar.). *Experimentations With Abstract Machines* [Online]. Available: https://github.com/elitheeli/oddities
[14] M. Cook, "Universality in elementary cellular automata," *Complex Syst.*, vol. 15, no. 1, pp. 1–40, 2004.
[15] J. Wolf, "OMG-WTF-PDF," in *Proc. 27th Chaos Comput. Congr.,* Dec. 2010.
[16] N. A. Danielsson, "Total parser combinators," in *Proc. 15th ACM SIGPLAN ICFP*, 2010, pp. 285–296.
[17] A. Koprowski and H. Binsztok, "TRX: A formally verified parser interpreter," in *Proc. Prog. Lang. Syst.*, LNCS 6012. 2010, pp. 345–365.
[18] T. Ridge, "Simple, functional, sound and complete parsing for all context-free grammars," submitted for publication.
[19] N. Chomsky, "On certain formal properties of grammars," *Inform. Comput./Inform. Control*, vol. 2, pp. 137–167, 1959.
[20] T. Berners-Lee and N. Mendelsohn. (2006). *The Rule of Least Power, Tag Finding* [Online]. Available: http://www.w3.org/2001/tag/doc/leastPower.html
[21] R. J. Hansen and M. L. Patterson, "Guns and butter: Toward formal axioms of input validation," in *Proc. Black Hat Briefings*, 2005.

[22] S. Ginsburg and S. Greibach, “Deterministic context free languages,” in *Proc. 6th Symp. Switching Circuit Theory Logical Design*, 1965, pp. 203–220.
[23] W. Cui, J. Kannan, and H. J. Wang, “Discoverer: Automatic protocol reverse engineering from network traces,” in *Proc. USENIX Sec. Symp.*, 2007.
[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol: HTTP/1.1*, Request for Comments: 2616, Jun. 1999.
[25] Information Sciences Institute, *Internet Protocol*, Request for Comments: 791, Sep. 1981.
[26] W. Ali, K. Sultana, and S. Pervez, “A study on visual programming extension of JavaScript,” *Int. J. Comput. Appl.*, vol. 17, no. 1, pp. 13–19, Mar. 2011.
[27] P. W. Abrahams, “A final solution to the dangling else of ALGOL 60 and related languages,” *Commun. ACM*, vol. 9, pp. 679–682, Sep. 1966.
[28] D. E. Knuth, “On the translation of languages from left to right,” *Inform. Control*, vol. 8, no. 6, pp. 607–639, 1965.
[29] H. J. S. Basten, “The usability of ambiguity detection methods for context-free grammars,” *Electron. Notes Theor. Comput. Sci.*, vol. 238, pp. 35–46, Oct. 2009.
[30] R. W. Floyd, “On ambiguity in phrase structure languages,” *Commun. ACM*, vol. 5, p. 526, Oct. 1962.
[31] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, Jul. 1948.
[32] W. Schramm, “How communication works,” in *The Process and Effects of Communication*. Champaign, IL: Univ. Illinois Press, 1954.
[33] D. K. Berlo, *The Process of Communication*. Concord, CA: Holt, Rinehart, and Winston, 1960.
[34] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commu. ACM*, vol. 12, no. 10, pp. 576–583, 1969.
[35] H. P. Grice, *Studies in the Way of Words*. Cambridge, MA: Harvard Univ. Press, 1989.
[36] B. Meyer, “Applying dsign by contract,” *Computer*, vol. 25, pp. 40–51, Oct. 1992.
[37] “CWE-77,” in *Common Weakness Enumeration*. Jul. 2008.
[38] T. Berners-Lee, R. Fielding, and L. Masinter, *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*, Request for Comments: 3986. Jan. 2005.
[39] D. Raggett, A. Le Hors, and I. Jacobs, *Forms in HTML Documents, HTML 4.01 Specification*, Dec. 1999.
[40] L. Carettoni and S. di Paola, *HTTP Parameter Pollution*. OWASP EU Poland, 2009.
[41] T. Pietraszek and C. V. Berghe, “Defending against injection attacks through context-sensitive string evaluation,” in *Proc. RAID*, 2005, pp. 124–145.
[42] W. G. J. Halfond, J. Viegas, and A. Orso, “A classification of SQL-injection attacks and countermeasures,” in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, Mar. 2006.
[43] rix. (2001, Aug.). Writing ia32 alphanumeric shellcodes. *Phrack* [Online]. *57(5)*. Available: http://www.phrack.com/issues.html?issue=57\&id=15
[44] Nergal. (2001, Dec.). The advanced return-into-lib(c) exploits: PaX case study. *Phrack* [Online]. *58(4)*. Available: http://www.phrack.com/issues.html?issue=58&id=4
[45] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” to be published.
[46] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proc. CCS*, 2007.
[47] T. Durden. (2002, Jul.). Bypassing PaX ASLR protection. *Phrack* [Online]. *59(9)*. Available: http://www.phrack.com/issues.html?issue=59\&id=9
[48] A. One. (1996, Aug.). Smashing the stack for fun and profit. *Phrack* [Online]. *49(14)*. Available: http://www.phrack.com/issues.html?issue=49\&id=14
[49] MaXX. Vudo malloc Tricks. *Phrack* [Online]. *57(8)*. Available: http://phrack.org/issues.html?issue=57&id=8
[50] Anonymous author. Once upon a free(). *Phrack* [Online]. *57(9)*. Available: http://phrack.org/issues.html?issue=57&id=9
[51] jp. (2003, Aug.). Advanced Doug Lea’s malloc exploits. *Phrack* [Online]. *61(6)*. Available: http://www.phrack.com/issues.html?issue=61\&id=6
[52] T. Newsham. (2000, Sep.). *Format String Attacks* [Online]. Available: http://www.thenewsh.com/∼newsham/format-string-attacks.pdf
[53] *National Vulnerability Database*, CVE-2005-3962, Dec. 2005.
[54] *National Vulnerability Database*, CVE-2011-1153, Mar. 2011.
[55] F. Valeur, D. Mutz, and G. Vigna, “A learning-based approach to the detection of SQL attacks,” in *Proc. DIMVA*, Jul. 2005, pp. 123–140.
[56] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” in *Proc. Int. Workshop Softw. Eng. Middleware*, 2005, pp. 106–113.
[57] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, “CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks,” *ACM Trans. Inf. Syst. Security*, vol. 13, no. 2, pp. 1–39, 2010.
[58] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” in *Proc. 33rd Symp. Principles Program. Lang.*, 2006, pp. 372–382.
[59] W. G. J. Halfond and A. Orso, “Preventing SQL injection attacks using AMNESIA,” in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 795–798.
[60] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, “Static checking of dynamically generated queries in database applications,” *J. ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, 2007.
[61] K. Wei, M. Muthuprasanna, and S. Kothari, “Preventing SQL injection attacks in stored procedures,” in *Proc. Aus. Softw. Eng. Conf.*, 2006, pp. 191–198.
[62] M. Muthuprasanna, K. Wei, and S. Kothari, “Eliminating SQL injection attacks: A transparent defense mechanism,” in *Proc. 8th IEEE Int. Symp. Web Site Evol.*, Sep. 2006, pp. 22–32.
[63] C. Gould, Z. Su, and P. Devanbu, “JDBC checker: A static analysis tool for SQL/JDBC applications,” in *Proc. Int. Conf. Soft. Eng.*, 2004, pp. 697–698.
[64] A. S. Christensen, A. Møller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in *Proc. 10th Int. Static Anal. Symp.*, 2003, pp. 1–18.
[65] S.-T. Sun and K. Beznosov, “Retrofitting existing web applications with effective dynamic protection against SQL injection attacks,” *Int. J. Secure Softw. Eng.*, vol. 1, pp. 20–40, Jan. 2010.
[66] W. G. J. Halfond and A. Orso, “AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks,” in *Proc. ASE 2005*, Nov. 2005, pp. 174–183.
[67] W. Halfond, A. Orso, and P. Manolios, “Using positive tainting and syntax-aware evaluation to counter SQL injection attacks,” in *Proc. FSE 2006*, Nov. 2006, pp. 175–185.
[68] P. Y. Gibello. (2002). *ZQL: A Java SQL Parser* [Online]. Available: http://zql.sourceforge.net/
[69] K. Kemalis and T. Tzouramanis, “SQL-IDS: A specification-based approach for SQL-injection detection,” in *Proc. Symp. Appl. Comput.*, 2008, pp. 2153–2158.
[70] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, “SQLProb: A proxy-based architecture toward preventing SQL injection attacks,” in *Proc. ACM Symp. Appl. Comput.*, 2009, pp. 2054–2061.
[71] *National Vulnerability Database*, CVE-2006-2313, May 2006.
[72] *National Vulnerability Database*, CVE-2006-2314, May 2006.
[73] D. Knuth, “Semantics of context-free languages,” *Math. Syst. Theory*, vol. 2, pp. 127–145, 1968.
[74] B. Kaliski. (1998, Mar.). *PKCS #1: RSA Encryption* [Online]. Available: http://tools.ietf.org/html/rfc2313
[75] B. Ford, “Parsing expression grammars: A recognition-based syntactic foundation,” in *Proc. 31st ACM SIGPLAN-SIGACT Symp. POPL*, 2004 pp. 111–122.
[76] M. Howard, J. Pincus, and J. Wing, “Measuring relative attack surfaces,” in *Computer Security in the 21st Century*, D. T. Lee, S. P. Shieh, and J. D. Tygar, Eds. New York: Springer, 2005, pp. 109–137.
[77] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation list (CRL) Profile*, RFC 5280, Obsoletes RFCs 3280, 4325, 4630, May 2008.
[78] P. Eckersley, “How unique is your web browser?,” Electronic Frontier Foundation, Tech. Rep., 2009.
[79] N. Mathewson and R. Dingledine, “Practical traffic analysis: Extending and resisting statistical disclosure,” in *Proc. PET Workshop*, LNCS 3424. May 2004, pp. 17–34.
[80] R. Clayton, “Failures in a hybrid content blocking system,” in *Proc. Fifth PET Workshop*, 2005, p. 1.
[81] F. Lindner, “The compromised observer effect,” *McAfee Security J.*, vol. 6, 2010.
[82] T. Ptacek, T. Newsham, and H. J. Simpson, “Insertion, evasion, and denial of service: Eluding network intrusion detection,” Secure Networks, Inc., West Palm Beach, FL, Tech. Rep., 1998.
[83] O. Arkin, “ICMP usage in scanning, the complete know-how,” Sys-Security Group, Tech. Rep., version 3.0, 2001.

[84] M Handley, V. Paxson, and C. Kreibich, “Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics,” in *Proc. 10th USENIX Security Symp.*, 2001, p. 9.
[85] U. Shankar and V. Paxson, “Active mapping: Resisting NIDS evasion without altering traffic,” in *Proc. IEEE Symp. Security Privacy*, May 2003, pp. 44–61.
[86] S. Siddharth. (2005, Dec.). *Evading NIDS, Revisited* [Online]. Available: http://www.symantec.com/connect/articles/evading-nids-revisited
[87] T. Garfinkel, “Traps and pitfalls: Practical problems in system call interposition based security tools,” in *Proc. Netw. Distributed Syst. Security Symp.*, Feb. 2003, pp. 163–176.
[88] S. A. Hofmeyr, A. Somayaji, and S. Forrest, “Intrusion detection system using sequences of system calls,” *J. Comput. Security*, vol. 6, no. 3, pp. 151–180, 1998.
[89] H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, “Anomaly detection using call stack information,” in *Proc. IEEE Symp. Security Privacy*, May 2003, p. 62.
[90] D. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems,” in *Proc. ACM Conf. CCS*, Nov. 2002, pp. 255–264.
[91] C. Taylor and C. Gates, “Challenging the anomaly detection paradigm: A provocative discussion,” in *Proc. 15th NSPW*, Sep. 2006, pp. 21–29.
[92] R. Sommer and V. Paxson, “Outside the closed world: On using machine learning for network intrusion detection,” in *Proc. IEEE Symp. Security Privacy*, May 2010, pp. 305–316.
[93] A. Somayaji, S. Hofmeyer, and S. Forrest, “Principles of a computer immune system,” in *Proc. NPSW*, 1998, pp. 75–82.
[94] A. Somayaji and S. Forrest, “Automated response using system-call delays,” in *Proc. 9th USENIX Security Symp.*, Aug. 2000.
[95] F. B. Schneider, “Enforceable security policies,” *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, Feb. 2000.
[96] K. Bhargavan, C. Fournet, and A. D. Gordon, “Modular verification of security protocol code by typing,” *SIGPLAN Not.*, vol. 45, 2010.
[97] S. Chaki and A. Datta, “ASPIER: An automated framework for verifying security protocol implementations,” in *Proc. CSF*, 2009, pp. 172–185.

Len Sassaman was a member of the COSIC Research Group, Katholieke Universiteit Leuven, Belgium. His early work with Cypherpunks on the Mixmaster anonymous remailer system and the Tor Project helped establish the field of anonymity research. In 2009, he and M. L. Patterson began formalizing the foundations of language-theoretic security, which he was involved with until the end of his life.

Dr. Sassaman passed away in July 2011. He was 31 years old.

Meredith L. Patterson lives in Brussels, Belgium.

As a Ph.D. student, she developed the first language-theoretic defense against SQL injection in 2005 and has continued expanding the technique since then. She is currently with Nuance Communications of Burlington, MA, USA. She is a Founder of Upstanding Hackers LLC, Cheyenne, WY, USA.

Sergey Bratus received the Ph.D. degree in mathematics from Northeastern University, Boston, MA.

He is currently a Research Assistant Professor of computer science with Dartmouth College, Hanover, NH. 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 was with BBN Technologies, Cambridge, MA, working on natural language processing research before coming to Dartmouth College.

Michael E. Locasto received the B.Sc. degree in computer science (*Magna Cum Laude*) from the College of New Jersey, Ewing, and the M.Sc. and Ph.D. degrees from Columbia University, New York.

He is currently an Assistant Professor with the Department of Computer Science, University of Calgary, Calgary, AB, Canada. He seeks to understand why it seems difficult to build secure, trustworthy systems, and how we can get better at it.