11-25-2011

Security Applications of Formal Language Theory

Len Sassaman
Dartmouth College

Meredith L. Patterson
Dartmouth College

Sergey Bratus
Dartmouth College

Michael E. Locasto
Dartmouth College

Anna Shubina
Dartmouth College

Security Applications of Formal Language Theory

Dartmouth Computer Science
Technical Report TR2011-709

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

November 25, 2011

Abstract

We present an approach to improving the security of complex, composed systems based on *formal language theory*, and show how this 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. These insights make possible future advances in software auditing techniques applicable to static and dynamic binary analysis, fuzzing, and general reverse-engineering and exploit development.

Our work provides a foundation for verifying critical implementation components with considerably less burden to developers than is offered by the current state of the art. It additionally offers a rich basis for further exploration in the areas of offensive analysis and, conversely, automated defense tools and techniques.

This report is divided into two parts. In Part I we address the formalisms and their applications; in Part II we discuss the general implications and recommendations for protocol and software design that follow from our formal analysis.

1 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.

For the traditional division of a system into hardware, firmware, and software, and of software into device drivers, generic OS kernel and its sub-layers, 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, *doing composition securely has emerged as the primary challenge to secure system construction.* Security practitioners know that interfaces are attack targets of choice, and that vulnerabilities are often caused by unexpected interactions of features across components; yet the reasons for this are elusive (except perhaps the attacker's desire for reliable execution of their exploits, which leads them to target the best-described parts of systems with tractable state; still, this does not explain our collective inability to design systems without unwanted feature interactions).

In this paper we argue that we need a new, stronger theoretic understanding of computational units' composition and of their underlying properties that make it empirically hard to "get it right", i.e., design systems without vulnerabilities caused by composition.

We show that there are strong computational-theoretic and formal language-theoretic reasons for the challenges of secure composition, and chart design principles to reduce these challenges. In particular, we show that the hard challenges of safe input handling and secure composition arise due to the underlying theoretically hard or unsolvable (i.e., UNDECIDABLE) problems that certain protocol designs and implementations essentially require to solve in order to secure them. We posit that the (unwitting) introduction of such problems in the 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 such designs 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 apparently 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.

Our argument focuses on the application of fundamental decidability results to the two basic challenges of composed and distributed system construction due to 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. *Multi-component 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 non-malicious 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 (cf. [1, 2, 3]), which our paper puts in solid theory perspective. We then elaborate the general principles of protocol design that follow from our analysis.

Structure of this paper

Our presentation consists of two parts. In *Part I* we make the case for formal language-theoretic approach to security, and show the direct relevance of various formalisms to practical, state-of-the-art classes of exploits and defences. In *Part II* we change tack and address protocols designers, developers, and security auditors with a set of recommendations derived from our formal analysis but formulated informally and accessibly. Readers interested in our recommendations may skip the formal arguments in Part I and go straight to Part II's for their summary.

We start with the motivation of our approach in Section 2 and review the necessary background formalisms in Section 3.

Then in Section 4 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 formal language properties of targeted data structures, which provides a novel and definitive way to analyze various suggested defences.

In Section 5 we show how to apply formal language-theoretic techniques to achieve rigorous, non-heuristic input validation. We start our discussion with SQL validation, but also show that the same approach applies to other context-free languages such as PKCS#1 (in Section 5.2 we prove that PKCS#1 is indeed context-sensitive). We also discuss flaws in previous validation approaches, and show why these flaws matter for practical security.

The discussion of flaws leads to us Section 6, 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 to enhance code auditing and protocol analysis.

In Section 7, 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 Part I with an outline of future work.

In Part II we recap the observations of Part I and formulate several principles that follow from our analysis in the preceding sections, and discuss their corollaries for designing and implementing protocols securely.

Part I. Security and Formal Language Theory

2 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 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 can be decided in a tractable fashion. We show that this requirement of being well-specified and tractably decidable is in fact a crucial pre-requisite of secure design and, in fact, its violation is the source of much of the 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 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 $\textsc{Equivalent}(\mathcal{L}(G) = \mathcal{L}(H))$ 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),

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

[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]).

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 [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.

Note. We must distinguish between *formal language theoretic security* — i.e., a security policy that governs inputs to systems and system components, restricting them to strings in the the formal languages particular to each component, and is enforced through the use of correct parsers for those languages as input handlers — and language-*based* security [19], which incorporates mechanisms that enforce security policies into the programming languages in which systems and their components are implemented. All of the vulnerabilities we will discuss could appear in programs written in languages that provide language-based security mechanisms. That said, language-based security mechanisms such as proof-carrying code and certifying compilers provide a means by which formal-language-theoretically secure implementations can be verified as such.

3 Background Formalisms

3.1 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 [20].

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. (E.g., 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 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] for that language, halts in an accepting state on all strings in the language, but

[3]Compare with the *decider*, a Turing machine that accepts strings in a *recursive* language (which is stronger than context-sensitive but weaker than recursively enumerable), rejects strings not in the language, and is guaranteed to halt.

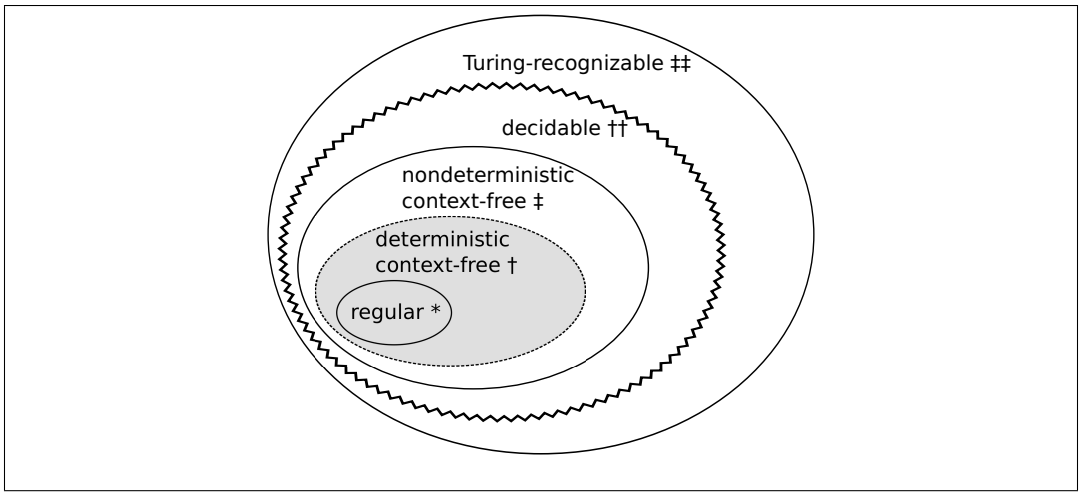

Figure 1: The Chomsky hierarchy of languages according to their expressive power. Languages correspond to grammars and automata as follows:
⋆ regular grammars, regular expressions, finite state machines
† unambiguous context-free grammars, deterministic pushdown automata
‡ ambiguous context-free grammars, non-deterministic 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.

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 [20, 10]. An unrestricted protocol grammar 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." [21]

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 two arbitrary context-free grammars, 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 [22]. The context-free grammars form two disjoint sets: deterministic and non-deterministic, corresponding to deterministic and non-deterministic pushdown automata respectively. All *unambiguous* context-free grammars are deterministic [23], 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 [24]. This applies to most network protocols and many file formats, where *length fields*, e.g. the `Content-Length` field of an HTTP header [25] or the `IHL` and `Length` fields of an IPv4 datagram [26], are commonplace. Programming language grammars that support statements of the form "`if B1 then if B2 then S1 else S2`", e.g. Javascript [27], are nondeterministic context-free (at best) due to the ambiguity the "dangling else" problem introduces [28]; if the shift-reduce conflict is resolved without adding braces or alternate syntax (e.g. `elif` or `end if`), the resulting grammar is non-context-free. Conveniently, the PostgreSQL, SQLite, and MySQL database engines all use LR grammars, which are deterministic context-free [29]. Membership tests for certain subclasses of LR (e.g. LALR, LR(k), etc.) and approximate ambiguity detection methods exist [30]; however, determining whether an arbitrary CFG is unambiguous is UNDECIDABLE [31].

3.2 Modeling Communication from a Security Standpoint

In 1948, Claude Shannon proposed a block-diagram model to describe systems which generate information at one point and reproduce it elsewhere [32]. 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 [33] and Berlo [34] 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 [35], 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 correctup to decidability [35]. When a destination program's semantics and implementation are provably correct, "whether or not it carries out its intended function" [35] 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 not infrequent 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 work. We are only interested in the semantics of $\mathcal{D}$ and $\mathcal{E}$.

4 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 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 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* [37] 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.

4.1 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

[4] This phenomenon suggests that Grice's Maxim of Relation [36] – "Be relevant" – applies to the pragmatics of artificial languages as well as natural ones.

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 [38].*

Example 2 (HTTP parameter pollution) *RFC 3986 [39] 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 [40] 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 behaviour for duplicate keys, in a query string or across input channels, can enable an attacker to override user-supplied data, control web application behaviour, and even bypass filters against other attacks [41].*

All types of injection leverage a weak boundary between control and data channels [42] to modify the structure, and thereby the execution semantics, of an input to an application component [22, 42]. Halfond et al [43] enumerate many heuristic injection defenses; in section 5.1 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*, which attempts to identify and reject malicious inputs; *programmatic abstraction*, which provides control channel access through an API and relegates user input to the data channel [43] 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.

4.2 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 [44], 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 [45] and its generalization, return-oriented programming [46, 47], 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 non-executable stacks [48].

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 [49] or a memory allocator's heap control structures [50, 51, 52]. 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 [53]. Other languages that support format strings exhibit similar vulnerabilities [54], and languages implemented in C, such as PHP, can succumb indirectly if unsafe input reaches a format string argument in the underlying implementation [55]. 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 [43] 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.

5 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.[5]

5.1 Injection Attacks and Context-Free Parse Tree Validation

In 2005, Dejector [22] presented a *context-free parse tree validation* approach to preventing SQLIA.[6] It introduced a formal construction for *restricted sublanguages* of SQL[7]; using this approach, validating an SQL query consists of testing it for membership in the sublanguage. Given a set of known-good queries and the formal grammar for the appropriate dialect of SQL, Dejector transforms the SQL grammar into a *subgrammar* that contains only the

[5] We consider the case of context-free languages such as SQL in this paper; our examination of the context-sensitive languages has been omitted for space, but is available in the extended version.

[6] The technique was independently discovered by a number of research teams. [56] and [57] published similar approaches around the same time; [58] 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 emphasis placed on the features which distinguish it from later attempts to implement these core ideas.

[7] Su and Wassermann [59] later independently arrived at the same construction.

rules required to produce exactly the queries in the known-good set[8]. 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 [59, 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 [56]) or approximate (as in [60] and [61]), 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:

5.1.1 Insufficiently strong automaton

Several automata-based validators [60, 61, 62, 63, 64] model the set of acceptable queries using a finite state machine, following the approach of Christensen et al. [65], 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 [66], 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" [43].

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 [67, 68], we question their efficacy against attacks that deliberately

[8] 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 in 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.

target the "impedance mismatch" between a generated FSA model and an underlying SQL grammar.

5.1.2 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 [57] compares parse trees of queries assembled with and without user input, using the ZQL parser [69]; CANDID [58] uses a "standard SQL parser based on SQL ANSI 92 standard, augmented with MySQL-specific language extensions"; SQLPrevent [66] uses ANSI SQL but does not mention which version. Others only state that the grammars they use are context-free [59, 70, 71].

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 [72, 73]. 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 6.

5.2 Parse Tree Validation in the Context-Sensitive Languages

In 2006, Daniel Bleichenbacher presented an RSA signature forgery attack against PKCS#1 implementations that do not correctly validate padding bytes [6]. We show that PKCS#1 is context-sensitive and can be validated in the same fashion as SQL, using an *attribute grammar* representation [74].

Theorem 1 *PKCS#1 is context-sensitive.*

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

Proof 2 *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 [75] 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 3 *We show that P is non-context-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. Q.E.D.

5.2.1 An attribute grammar for PKCS#1

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(⟨Hash⟩$_2$) + 1

6 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 non-malicious information exchange, prior security assumptions may fail. In order to retain the security properties of each process in an environment where several processes participate in sequentialized communication, one must show that the properties also hold for the concurrent system they constitute.

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. (Section 3.2), the goal of a parse tree differential attack is to find combinations of M_{src}, $\mathcal{E}_{src}$, and $\mathcal{D}_{dst}$ 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]).

6.1 Attack Surface Discovery

We have used edge-cases identified by parse tree differential analysis to isolate serious vulnerabilities in X.509. 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`*!*

6.2 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 [77]. 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 [78]. 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 [79]. 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 non-interdicted 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, BT's, 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".

6.2.1 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 $\rightarrow$ 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.

7 Why Johnny Can't Detect

One arguably non-principled 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, e.g., [80]).

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 [81] 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., [82] 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., [83, 84, 85]).

In host-based intrusion detection systems (HIPS), the problem of matching the "defending" computation with the targeted computation is no less pronounced. For example, Garfinkel [86] 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 coule 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 [87, 88]. 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 [89]; 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 [90] 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 [91]. 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. [92, 87, 93]. 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 [94] 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.

8 Future Work

In an expanded version of this paper, we will show how context-free parse tree validation can be extended to the context-sensitive languages, using *attribute grammars* [74] as both a representational formalism and a code-generation tool [95].

Our future work will integrate existing work on generation of verifiable, guaranteed-terminating parsers [16, 17, 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.

Part II. Language-theoretic recommendations for secure design

9 A summary of language-theoretic view of security

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.*

Indeed, a system's security is largely defined by what computations can and cannot occur in it under all possible inputs. The parts of a system where input-driven computation occurs are typically meant to act together as a *recognizer* for the inputs' validity, i.e., they are expected to accept valid or expected inputs,[9] and reject bad inputs. Exploitation — an *unexpected* input-driven computation — usually occurs there as well; thinking of it as an input language *recognizer bug* often helps find exploitable vulnerabilities.

Crucially, for complex inputs (input languages), the recognizer 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 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.

Furthermore, most approaches to *input validation* employ hand-written recognizers, rather than recognizers generated from a protocol's grammar even if such a formal grammar is available. These ad-hoc 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 that allow for recursively nested data structures, such as context-free languages, which require more powerful recognizers.

A protocol that appears to frustratingly resist efforts to implement it securely, to weed out vulnerabilities by comprehensively testing its implementations, or 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) UNDECIDABLE problems. Of course, for such problems no "80/20" engineering approximation is possible.

Conversely, as Sassaman and Patterson have shown, 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 considerably eases the task of vulnerability hunting and exploit construction, as it helps to find false data and state validity assumptions that open ways to manipulate the target program's state.

Recognizers and "weird machines". The latter observation perfectly agrees with the modern understanding of exploit programming as "setting up, instantiating, and programming a weird machine" [98, 99]. The term "weird machine" refers to the computational environment (embedded in the target system) that consists of a subset of the actually possible system states (as opposed to valid states envisioned by designers and programmers), which, as Dullien has observed, explodes in the presence of errors and bugs, and of transi-

[9] See our discussion of Postel's Principle for distinguishing between valid and expected inputs.

tions between them caused by crafted inputs. In a word, malicious computation constructed by the attacker runs on the "weird machine" inside the target.

Thus programming an exploit involves enumerating the relevant unanticipated system states and transitions (cf. the axiomatic definition of vulnerability by Bishop et al. [100] in terms of unauthorized system states and transitions). Failures of input recognition, and in particular false assumptions regarding data received as input provide a rich source of such states and transitions.

For example, extra state typically arises when the programmer incorrectly believes certain constraints on input-derived data to be satisfied (i.e., enforced by input validation logic), and programs to that assumption, actually creating implicit data flows and unexpected control flows that contribute to the "weird machine".

10 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 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 deterministic 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, and
- 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 an input language should be regarded as a grant of additional privilege, and thus increasing security risks. Such increases should therefore be entered into reluctantly, with eyes open, and should be considered as part of a formal risk assessment. In the very least, designers should be guided by the Chomsky hierarchy (described in the sidebar).

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 any 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+CSS 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.

The example of X.509 implementations ([101]) shows that this problem is directly related to the insecurity of distributed systems' tasks. Moreover, undecidability essentially precludes construction of efficient code testing and algorithmic verification 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" [1] 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 "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 the 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. We argue in an upcoming article [102] that a strict reading of the Postel's Principle should in fact discourage ambiguity and computational complexity in protocols.

10.1 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 that need a Nondeterministic Pushdown Automaton to recognize 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.

10.2 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 an 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 an 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. [103]).

11 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 aimed to address the security risks of an input language in common Internet use is *Blitzableiter* by Felix 'FX' Lindner and Recurity Labs [80]. 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 attackers construct malicious computations.

We note that that the approach of normalizing complex network protocols to remove both ambiguity and contsructs requiring excessive computation power (e.g., [83]) is a close correlate, and works for the same reason.

Another interesting example are the observations by D.J. Bernstein on the 10 years of *qmail* [3] 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, pointing to hand-crafting input handling code for efficiency as a dangerous distraction, and his stress on using UNIX context isolation primitives as a way of enforcing *explicit data flows* (thus hobbling construction of malicious computations). 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 see Bernstein's insights as being actually in line with our proposed update.

12 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 inputs language(s) must be kept at a minimum when designing protocols. This serves 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 priciples that could make protection from unexpected computations a reality, if followed in 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.

13 Acknowledgments

The authors would like to thank Ed Feustel for his observations on the security of composed systems; Dan Kaminsky, for his collaboration in the analysis of flaws in ASN.1 parsers; Andreas Bogk, Rik Farrow, Dan McCardle, James Oakley, Frank Piessens, and Sean W. Smith for helpful suggestions on earlier versions of this paper, and Nikita Borisov, Nessim Kisserli, Felix Lindner, Doug McIlroy, and David Molnar, for their insightful conversations during the process of this research.

The work of Len Sassaman was supported in part by the Research Council K.U.Leuven: GOA TENSE (GOA/11/007), and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

[1] Dan Geer, "Vulnerable Compliance", *;login: The USENIX Magazine*, vol. 35, no. 6, December 2010.

[2] Felix 'FX' Lindner, "The Compromised Observer Effect", *McAfee Security Journal*, vol. 6, 2010, Dan Sommer (Editor).

[3] Daniel J. Bernstein, "Some Thoughts on Security after Ten Years of qmail 1.0", in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, New York, NY, USA, 2007, CSAW '07, pp. 1–10, ACM.

[4] Leslie Lamport, "Proving the Correctness of Multiprocess Programs", *IEEE Transactions on Software Engineering*, vol. 3, pp. 125–143, 1977.

[5] Ranjit Jhala and Rupak 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] Tetsuya Izu, Takeshi Shimoyama, and Masahiko Takenaka, "Extending Bleichenbacher's Forgery Attack", *J. Information Processing*, vol. 16, pp. 122–129, September 2008.

[8] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure", in *Financial Cryptography*. 2010, pp. 289–303, Springer.

[9] Géraud Sénizergues, "L(A)=L(B)? Decidability Results from Complete Formal Systems", *Theor. Comput. Sci.*, vol. 251, no. 1-2, pp. 1–166, 2001.

[10] Michael Sipser, *Introduction to the Theory of Computation, Second Edition, International Edition*, Thompson Course Technology, 2006.

[11] Stephan Kepser, "A Simple Proof for the Turing-Completeness of XSLT and XQuery", in *Extreme Markup Languages*, 2004.

[12] Ruhsan Onder and Zeki Bayram, "XSLT Version 2.0 Is Turing-Complete: A Purely Transformation Based Proof", in *Implementation and Application of Automata*, Oscar Ibarra and Hsu-Chun Yen, Eds. 2006, vol. 4094 of *Lecture Notes in Computer Science*, pp. 275–276, Springer Berlin / Heidelberg.

[13] Eli Fox-Epstein, "Experimentations with Abstract Machines", https://github.com/elitheeli/oddities, March 2011.

[14] Mathew Cook, "Universality in Elementary Cellular Automata", *Complex Systems*, vol. 15, no. 1, pp. 1–40, 2004.

[15] Julia Wolf, "OMG-WTF-PDF", 27th Chaos Computer Congress, December 2010.

[16] Nils Anders Danielsson, "Total Parser Combinators", in *Proc. 15th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2010, ICFP '10, pp. 285–296.

[17] A. Koprowski and H. Binsztok, "TRX: A Formally Verified Parser Interpreter", in *Prog. Languages and Systems*, vol. 6012 of *LNCS*, pp. 345–365. Springer, 2010.

[18] Tom Ridge, "Simple, Functional, Sound and Complete Parsing for All Context-free Grammars", Submitted for publication, 2011.

[19] F.B. Schneider, J.G. Morrisett, and R. Harper, "A Language-Based Approach to Security", in *Informatics - 10 Years Back. 10 Years Ahead.* 2001, Springer-Verlag.

[20] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Computation/information and Control*, vol. 2, pp. 137–167, 1959.

[21] Tim Berners-Lee and Noah Mendelsohn, "The Rule of Least Power", Tag finding, W3C, 2006, http://www.w3.org/2001/tag/doc/leastPower.html.

[22] Robert J. Hansen and Meredith L. Patterson, "Guns and Butter: Toward Formal Axioms of Input Validation", *Proceedings of the Black Hat Briefings*, 2005.

[23] Seymour Ginsburg and Sheila Greibach, "Deterministic Context Free Languages", in *Proc. 6th Symp. Switching Circuit Theory and Logical Design*, 1965, pp. 203–220.

[24] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang, "Discoverer: Automatic Protocol Reverse Engineering from Network Traces", in *USENIX Sec. Symp.*, 2007.

[25] 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, June 1999.

[26] Information Sciences Institute, "Internet Protocol", Request for Comments: 791, Sept. 1981.

[27] Wajid Ali, Kanwal Sultana, and Sophia Pervez, "A Study on Visual Programming Extension of JavaScript", *Int'l J. Computer Applications*, vol. 17, no. 1, pp. 13–19, March 2011.

[28] Paul W. Abrahams, "A Final Solution to the Dangling else of ALGOL 60 and Related Languages", *Commun. ACM*, vol. 9, pp. 679–682, September 1966.

[29] Donald E. Knuth, "On the Translation of Languages from Left to Right", *Information and Control*, vol. 8, no. 6, pp. 607 – 639, 1965.

[30] H. J. S. Basten, "The Usability of Ambiguity Detection Methods for Context-Free Grammars", *Electron. Notes Theor. Comput. Sci.*, vol. 238, pp. 35–46, October 2009.

[31] Robert W. Floyd, "On Ambiguity in Phrase Structure Languages", *Commun. ACM*, vol. 5, pp. 526–, October 1962.

[32] Claude E. Shannon, "A Mathematical Theory of Communication", *The Bell system technical journal*, vol. 27, pp. 379–423, July 1948.

[33] Wilbur Schramm, *The Process and Effects of Communication*, chapter "How Communication Works", University of Illinois Press, 1954.

[34] D.K. Berlo, *The Process of Communication*, Holt, Rinehart, and Winston, 1960.

[35] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.

[36] H. P. Grice, *Studies in the Way of Words*, Harvard University Press, 1989.

[37] B. Meyer, "Applying "Design by Contract"", *Computer*, vol. 25, pp. 40–51, October 1992.

[38] "CWE-77", Common Weakness Enumeration, July 2008.

[39] Tim Berners-Lee, Roy Fielding, and Larry Masinter, "RFC 3986, Uniform Resource Identifier (URI): Generic Syntax", Request for Comments: 3986, Jan 2005.

[40] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, "Forms in HTML documents", HTML 4.01 Specification, December 1999.

[41] Luca Carettoni and Stefano di Paola, "HTTP Parameter Pollution", OWASP EU Poland, 2009.

[42] Tadeusz Pietraszek and Chris Vanden Berghe, "Defending Against Injection Attacks Through Context-Sensitive String Evaluation", in *RAID*, 2005, pp. 124–145.

[43] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL-Injection Attacks and Countermeasures", in *Proc. of the IEEE Int'l Symposium on Secure Software Engineering*, March 2006.

[44] rix, "Writing ia32 Alphanumeric Shellcodes", Phrack 57, August 2001.

[45] Nergal, "The Advanced return-into-lib(c) Exploits: PaX case study", Phrack 58, Dec 2001.

[46] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage, "Return-Oriented Programming: Systems, Languages, and Applications", 2009, In review.

[47] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", in *Proceedings of CCS*, 2007.

[48] Tyler Durden, "Bypassing PaX ASLR protection", Phrack 59, Jul 2002.

[49] Aleph One, "Smashing the Stack for Fun and Profit", Phrack 49, Aug 1996.

[50] MaXX, "Vudo malloc Tricks", Phrack 57:8, `http://phrack.org/issues.html?issue=57&id=8`.

[51] anonymous author, "Once upon a free()", Phrack 57:9, `http://phrack.org/issues.html?issue=57&id=9`.

[52] jp, "Advanced Doug Lea's malloc Exploits", *Phrack Magazine*, vol. 61, no. 6, Aug 2003.

[53] Tim Newsham, "Format String Attacks", http://www.thenewsh.com/ newsham/format-string-attacks.pdf, Sep 2000.

[54] "CVE-2005-3962", National Vulnerability Database, December 2005.

[55] "CVE-2011-1153", National Vulnerability Database, March 2011.

[56] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks", in *Proc. DIMVA*, July 2005, pp. 123–140.

[57] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti, "Using Parse tree Validation to Prevent SQL Injection Attacks", in *Proc. Int'l Workshop on Software Engineering and Middleware*, 2005, pp. 106–113.

[58] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks", *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, pp. 1–39, 2010.

[59] Zhendong Su and Gary Wassermann, "The Essence of Command Injection Attacks in Web Applications", in *33rd Symp. Princ. of Prog. Langs*, 2006, pp. 372–382.

[60] William G. J. Halfond and Alessandro Orso, "Preventing SQL Injection Attacks Using AMNESIA", in *Proc. 28th Int'l Conf. on Software engineering*, 2006, pp. 795–798.

[61] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications", 2007.

[62] Ke Wei, M. Muthuprasanna, and Suraj Kothari, "Preventing SQL Injection Attacks in Stored Procedures", in *Proc. Aus. Software Eng. Conf.*, 2006, pp. 191–198.

[63] M. Muthuprasanna, Ke Wei, and Suraj Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism", in *Proc. 8th IEEE Int'l Symposium on Web Site Evolution*, 2006, pp. 22–32.

[64] Carl Gould, Zhendong Su, and Premkumar Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications", in *Int'l Conf. Soft. Eng.*, 2004, pp. 697–698.

[65] A.S. Christensen, A. Møller, and M.I. Schwartzbach, "Precise Analysis of String Expressions", in *Proc. 10th Int'l Static Analysis Symp.*, 2003, pp. 1–18.

[66] San-Tsai Sun and Konstantin Beznosov, "Retrofitting Existing Web Applications with Effective Dynamic Protection Against SQL Injection Attacks", *Int'l J. Secure Software Engineering*, vol. 1, pp. 20–40, Jan 2010.

[67] W.G.J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks", in *ASE 2005*, Long Beach, CA, USA, Nov 2005.

[68] W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks", in *FSE 2006*, Nov 2006, pp. 175–185.

[69] P.Y. Gibello, "ZQL: a Java SQL parser", http://zql.sourceforge.net/, 2002.

[70] Konstantinos Kemalis and Theodores Tzouramanis, "SQL-IDS: a Specification-based Approach for SQL-injection Detection", in *Proc. Symposium on Applied Computing*, 2008, pp. 2153–2158.

[71] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou, "SQLProb: a Proxy-based Architecture towards Preventing SQL Injection Attacks", in *ACM Symposium on Applied Computing*, 2009, pp. 2054–2061.

[72] "CVE-2006-2313", National Vulnerability Database, May 2006.

[73] "CVE-2006-2314", National Vulnerability Database, May 2006.

[74] Donald Knuth, "Semantics of Context-Free Languages", *Mathematical Systems Theory*, vol. 2, pp. 127–145, 1968.

[75] B. Kaliski, "PKCS #1: RSA Encryption", March 1998, `http://tools.ietf.org/html/rfc2313`.

[76] Michael Howard, Jon Pincus, and Jeannette Wing, "Measuring Relative Attack Surfaces", in *Computer Security in the 21st Century*, D. T. Lee, S. P. Shieh, and J. D. Tygar, Eds., pp. 109–137. Springer US, 2005.

[77] Peter Eckersley, "How Unique Is Your Web Browser?", Tech. Rep., Electronic Frontier Foundation, 2009.

[78] Nick Mathewson and Roger Dingledine, "Practical Traffic Analysis: Extending and Resisting Statistical Disclosure", in *Proceedings of Privacy Enhancing Technologies Workshop (PET 2004)*, May 2004, vol. 3424 of *LNCS*, pp. 17–34.

[79] Richard Clayton, "Failures in a Hybrid Content Blocking System", in *Fifth Privacy Enhancing Technologies Workshop, PET 2005*, 2005, p. 1.

[80] Felix Lindner, "The Compromised Observer Effect", *McAfee Security Journal*, vol. 6, 2010, Dan Sommer (Editor).

[81] Thomas Ptacek, Timothy Newsham, and Homer J. Simpson, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection", Tech. Rep., Secure Networks, Inc., 1998.

[82] Ofir Arkin, "ICMP Usage in Scanning, The Complete Know-how", Tech. Rep., The Sys-Security Group, 2001, version 3.0.

[83] Mark Handley, Vern Paxson, and Christian Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics", in *Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001, SSYM'01, pp. 9–9, USENIX Association.

[84] Umesh Shankar and Vern Paxson, "Active Mapping: Resisting NIDS Evasion without Altering Traffic", in *IEEE Symposium on Security and Privacy*, 2003, pp. 44–61.

[85] Sumit Siddharth, "Evading NIDS, revisited", `http://www.symantec.com/connect/articles/evading-nids-revisited`, December 2005, (updated Nov 2010).

[86] Tal Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools", in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[87] S. A. Hofmeyr, Anil Somayaji, and S. Forrest, "Intrusion Detection System Using Sequences of System Calls", *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[88] Henry H. Feng, Oleg Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong, "Anomaly Detection Using Call Stack Information", in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[89] David Wagner and Paolo Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems", in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2002.

[90] Carol Taylor and Carrie Gates, "Challenging the Anomaly Detection Paradigm: A Provocative Discussion", in *Proceedings of the* 15^{th} *New Security Paradigms Workshop (NSPW)*, September 2006, pp. 21–29.

[91] R. Sommer and V. Paxson, "Outside the Closed World: on Using Machine Learning for Network Intrusion Detection", in *Proc. IEEE Symposium on Security and Privacy*, May 2010.

[92] A. Somayaji, S. Hofmeyer, and S. Forrest, "Principles of a Computer Immune System", in *Proceedings of the New Security Paradigms Workshop (NPSW)*, 1998, pp. 75–82.

[93] A. Somayaji and S. Forrest, "Automated Response Using System-call Delays", in *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[94] Fred B. Schneider, "Enforceable Security Policies", *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, February 2000.

[95] W. Swierstra, "Why Attribute Grammars Matter", The Monad Reader, July 2005.

[96] K. Bhargavan, C. Fournet, and A.D. Gordon, "Modular Verification of Security Protocol Code by Typing", *Sigplan Notices*, vol. 45, 2010.

[97] S. Chaki and A. Datta, "ASPIER: An Automated Framework for Verifying Security Protocol Implementations.", in *CSF*, 2009, pp. 172–185.

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

[99] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina, "Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation", *;login:*, December 2011.

[100] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies", Tech. Rep. Technical Report CSE-96-11, Dept. of Computer Science, University of California at Davis, Davis, CA 95616-8562, September 1996.

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

[102] Len Sassaman, Meredith L. Patterson, and Sergey Bratus, "A Patch for Postel's Robustness Principle", *submitted to IEEE Security and Privacy Journal*, 2011.

[103] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith, "New Directions for Hardware-assisted Trusted Computing Policies (Position Paper)", 2008.