Interview with Sergey Bratus

RIK FARROW

Sergey Bratus is a Research Associate Professor of Computer Science at Dartmouth College. He helped co-found the LangSec movement and is interested in understanding and mitigating unintended computation. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. sergey@cs.dartmouth.edu

Rik is the editor of *;login:*. rik@usenix.org

Disclaimer: The views presented in this interview are the author's personal views and do not necessarily represent the views of the U.S. Federal Government or its components, which partially funded some of the research presented at the LangSec workshop.

I first met Sergey Bratus during the USENIX Security Symposium in 2011. Sergey caught up to me in a stairwell at the Sir Francis Drake Hotel in San Francisco and started to make a pitch about something I had never heard of before. LangSec, short for Language Security, is a different way of thinking about both how to program more securely and why software gets exploited.

I found myself immediately intrigued, and Sergey has co-authored several articles and papers related to LangSec over the years. He also co-founded a LangSec workshop with Meredith Patterson, co-located with IEEE Security and Privacy ("Oakland") [1]. When I was studying papers at USENIX Security '20, I noticed several that appeared to have strong tie-ins to LangSec and decided to invite Sergey for an interview.

Rik Farrow: Software gets hacked when presented with input that manipulates the software in unexpected ways. I recall from early LangSec articles that any input parser that is more complex than a pushdown automaton will be vulnerable to this type of hacking. Do I have this right, and why are more complex parsers vulnerable?

Sergey Bratus: The programmer who sits down to write a parser faces a task quite unlike any other engineering task. All other kinds of engineers design for some well-defined operating environment conditions: this much wind speed for a bridge, this much current for an electric circuit, this expected temperature interval for a chip, etc. Within these conditions, the design must behave predictably: safety comes from predictability. By contrast, input-taking software, i.e., its parser, is supposed to withstand *any* inputs at all, an operating environment that cannot be easily searched or simulated. Yet, as with any other engineering, safety only comes from predictability.

Thus safety of a parser critically depends on the ability of the programmer to correctly predict the parser's behavior on all possible inputs. This is really hard, because reasoning about program behaviors *in general* is hard (or even algorithmically impossible) and is only feasible when the programmer walks a fairly narrow path, by correctly implementing automata that we *can* reason about and assuming no more about the inputs than these automata (if correctly implemented) can check.

Pushdown automata and their corresponding context-free languages are one particular sweet spot of predictability for which we have the mathematical and computing means of automated reasoning. This sweet spot is really something of a mathematical miracle, given how hard the general problem is.

In a word, every parser implemented on a general-purpose ISA wants to be a virtual machine on its inputs that matches the computing power of that ISA. Restraining it from being that machine for the attacker is what LangSec is about; it is surprising and fascinating that it is possible and practical to do so.

Various caveats apply, which LangSec aims to address in ways practical for a programmer who is not looking to be a mathematician or formal language theorist. However, the thing that makes it at all possible is the language-based approach, which gives us just the predictability, that is, safety, properties that we can check for and that aren't hard to express and understand.

Surprisingly, as the recent workshop's morning keynote [1] David Walker argued, this is also true for predicting behaviors of not just parsers but also networks. So the surprising effectiveness of using language-based models of computing system behaviors extends beyond what we normally think of as parsers.

RF: Programmers often build parsers according to their reading of a protocol specification. An infamous example of this going wrong was Heartbleed, where the TLS protocol included two different length values, and a popular implementation checked one while using the other. At USENIX Security, the "Composition Kills" paper [2] examines how the intersection of three email sender authentication protocols—SPF, DKIM, and DMARC—actually fail to authenticate the sender. Are protocols part of the problem that LangSec addresses?

SB: Yes. From its inception [17], LangSec has been calling attention not only to unintended behaviors of inputs on parsers, but also to security consequences of *parser differentials*, that is, divergent interpretations of the same messages by different parsers.

To have any predictability in a distributed system—which is really just a fancy name for a system with more than one component—it is natural to implicitly assume that all of its parsers interpret messages passed between the components in the same way. Whenever this assumption, made explicitly or implicitly, is violated, vulnerability likely ensues.

Vulnerabilities with the root cause in parser differentials have been in the news lately. The HTTP Desync vulnerabilities [3] such as the F5 Big-IP vulnerability [4], the "Psychic Paper" vulnerability in MacOS [5], and a vulnerability in GitLab [6] all involve parser differentials. Major past examples include several Android Master Key vulnerabilities [7], a timeless classic.

LangSec's perspective on the insecurity potential of parser differentials has been getting some notice. Another notable, recently published academic paper [8] discusses parsing of standard protocols and refers to LangSec. Dave Aitel drew attention to the LangSec nature of this growing vulnerability class on his DailyDave mailing list (https://seclists.org/dailydave/2020/q3/9). To quote Dave:

> Ten years ago a lot of the security community had a discussion about "LangSec"...which turns out to have been entirely correct in retrospect....
>
> Most people look at HTTP Desync as simply using Content-Length confusion—figuring out ways to make one request look like it's not the same length, and using that for SSRF or XSS or various other attacks. But *ANY DIFFERENCE IN THE PARSERS* leads to critical level attacks.

The surface of LangSec analysis in distributed systems has only been scratched, so there are likely many more major vulnerabilities waiting to be discovered.

RF: LangSec seems to be heading in the direction of language-based designs, that is, requiring language to provide security assurances. Java was supposed to do this, but there are many Java exploits. Some exist because there are extensions to Java written in unsafe languages, like C. But I believe that people have exploited Java via the bytecode itself.

SB: LangSec targets the root causes of insecurity on a different level than efforts aimed at general-purpose programming languages.

Java and other memory-safe languages target the ability of the programmer to unwittingly (or deliberately) create memory corruption or (non-corrupting) type confusion. For Java and JavaScript, this ability was largely taken away from the developer, which is a net positive, but not a panacea.

The problem of unexpected and unchecked input remains. Now these inputs are stored in memory-safe ways, but they are still not what the processing code expects, and they are still acted on. There is a lot of room in a general-purpose language to go wrong when acting on data that's not what the programmer expects. For programs such as web apps that produce outputs and issue commands, this problem will manifest as either the outputs or the commands not being as expected.

LangSec, by contrast, aims to offer general solutions that focus first and foremost on data languages, also called data formats.

Without a clear understanding of input and output data languages involved in a task, the programming language is only exchanging one bug class for another. For example, Java and JavaScript made memory corruption harder, although, as you note, not impossible. Still, regular programmers cannot accidentally corrupt memory with their code alone: it has to come from flaws in the language runtime implementation or, more typically, from their interactions. However, complexities of data languages and their transformations immediately manifested themselves in XSS, command execution bugs, parser differentials, etc., making notionally memory-safe web apps notoriously vulnerable to an array of attacks much less sophisticated than memory corruption exploits.

Note that outputs and the code that creates them ("unparsers") are as important as the inputs and their handling code: see, for example, [9] and the first workshop paper [1].

My understanding is that Google and Facebook had to integrate intricate type systems with their web development tool chains to just keep a lid on this problem, and their solutions are specialized to their respective processes.

LangSec absolutely takes to heart the dictum of functional programming: "Make illegal state unrepresentable." This dictum calls on a language designer or an API architect to construct the language or the API so as to make it impossible for the programmer to create illegal state—at least not without the compiler complaining very loudly. However, properly implementing this dictum wherever inputs or outputs are involved requires understanding what are the legal and illegal states of input, and the same for output. It requires LangSec.

RF: When I interviewed Natalie Silvanovich [10], she seemed to conflate the use of dynamic languages (those that handle memory allocation and freeing dynamically, like Rust and Go) as part of LangSec. What do you think?

SB: I'd like to start by saying that LangSec greatly benefited from interest and feedback from extraordinary vulnerability researchers, who were, in fact, among the first to grasp its practical value. For example, the closing keynote of the first LangSec workshop was by Felix "FX" Lindner, an early supporter of LangSec. This makes perfect sense, because leading vulnerability researchers see general patterns of software weaknesses, of input-driven exploitation, and of how its non-systematic mitigations fail. LangSec offered a unified and actionable way of explaining these patterns, and top vulnerability researchers were among the first to appreciate it.

In your interview, Natalie's take on the nature and scope of LangSec is spot-on:

> [LangSec] views the root cause of security issues to be that most protocols and other input formats are poorly defined and often have many undefined states, and the programming languages that process them also support a huge amount of undefined behavior. [LangSec] thinks all software should abstract out all input processing code, and design and implement it in a way that is verifiable, and has no undefined states or behavior.

As I mentioned earlier, and as Natalie notes, the common idea of managed-memory languages is to make illegal memory states impossible for the programmer to unwittingly create while writing regular code. Notably, LangSec aims further than basic memory corruption. Indeed, there are numerous examples of memory-safe software with deep flaws due to ad hoc handling of its input and output languages.

However, Natalie raised another important point in that interview: there are and will be bugs in programming languages and environments intended to be memory-safe or otherwise offer safety assurances. In this year's LangSec workshop's amazing invited talk, Natalie connected this insight with specific features of JavaScript that have been causing huge headaches worldwide, given how JavaScript has been "eating the Internet"—and pinpointed the ways out. See her slides at [1] for the discussion of these troublesome features. Natalie has a wonderful intuition here, which is entirely LangSec but takes us beyond file and message formats.

I would describe it as follows: Natalie sees data structures allocated in memory as data languages, with the runtime memory management code servicing these structures as parsers. Programming language feature choices made by JavaScript or Go about what kinds of objects and how their relationships are representable in the language force the implementations of these languages to handle ever more complex data languages of bytes in memory: for example, on the heap. Consequently, unnecessary complexity of these features causes the same devastating effects as unnecessary format complexity does on the software that processes the formats.

Any piece of the language's native runtime, including the memory manager and garbage collector, parses memory bytes all the time and often must decide if a chunk it parses is valid or not before it acts. Moreover, advanced memory management means that multiple actors read and write memory concurrently, and their parsing actions must all be synchronized, or else corruption occurs. There is a rich literature of hacker research here, including many nifty attacks on browsers and OS kernels. This area is waiting to be explored from the LangSec perspective, and Natalie's invited talk pointed out a very rich example.

RF: You've mentioned that language-based approaches could turn out to be amazingly productive in understanding routing. Can you explain how LangSec intersects with network routing?

SB: Routing and other network-processing tasks must process streams of packets or, at a higher level, events. These packets or events change the internal state of the receiving program. Essentially, just like a parser, a network stack or function performs input-driven computation. Many questions about routing come down to modeling and understanding this computation, and assuring that it is safe—that is, behaves predictably for all inputs it might receive.

With modern verification tools we can try to prove that a distributed system has some desired behavioral properties. But which properties and models are tractable to explore?

It turns out that thinking about sequences of networking events as a data language that drives language-processing tasks is surprisingly productive for reasoning about and verifying network router behaviors. Not only that, but understanding the routers' many configuration options as dialects of a common language was also an efficient way of organizing and searching the space of diverse configurations. The latter is arguably less surprising, because human designers of these spaces, as all humans, are creatures of language and tend to implicitly impose language-based ordering on complex spaces.

This was the subject of this year's workshop's morning keynote by Princeton's David Walker [1]. Of course, as the original LangSec paper [11] points out, treating observable system and network events as streams processed by input-driven automata predates LangSec. For example, Fred Schneider used this approach to characterize classes of enforceable security policies [12] and cited Lamport's prior work. However, it's still fascinating that formal language-based approaches are so productive far beyond parsing.

RF: Forms of distributed computing, such as cloud functions, are growing in popularity today. Cloud functions use RPCs and queues to communicate, and that seems to me to be an opportunity to either make things better by observing LangSec or much worse through the use of ambiguous protocols. Would you comment on that?

SB: This is very much the case: there is both the opportunity and the danger.

The danger is already manifesting itself in the surge of high-impact parser differential bugs. Recall Dave Aitel's quote above. Note that we don't yet have effective ways of fuzzing for parser differentials. So we are in a much worse position with respect to parser differential bugs than we are with regard to memory corruption bugs, where coverage-driven fuzzing in combination with various sanitizers have gotten really good.

There is also the opportunity. Exposing interfaces without the false comfort of keeping them "private" and only receiving well-formed data or only data from one particular writer applies evolutionary pressure towards properly defining these interfaces. LangSec is there as a natural match for this problem.

The story of the Amazon API Mandate as told by Steve Yegge [13] is the story of such evolutionary pressure creating a qualitatively better platform. From the LangSec perspective, this story is not surprising—it is an iconic story of the correct intuition.

RPC messages are explicitly data languages, and open cloud environments will exert pressure to validate RPC messages before acting on them. However, it is important to get the design of these data languages right, so that validating these inputs doesn't grow into intractable problems we encounter with legacy formats.

As cloud systems grow rapidly, so could their technical debt. For example, for many application protocols, their expressions in Protocol Buffers happen to be the closest they ever got to a mechanized specification. However, these specifications themselves may be ambiguous and vulnerable to parser differentials. Critiques such as [14] strongly urge caution.

These problems are going to be very important as we move to serverless styles of programming (AWS Lambda and Fargate, Azure Functions, etc.). They will take a while to explore and understand, just like understanding the significance of parser differentials took almost a decade, but to avoid accumulating insurmountable amounts of technical debt, we should start now.

RF: The Rust programming language claims to offer unprecedented security assurances in systems programming. Rust's secret weapon appears to be lightweight memory safety through compiler-imposed isolation, instead of having to rely on much more expensive safety solutions such as separating memory contexts with x86 hardware privilege rings or automatic memory management. Will LangSec remain relevant if Rust becomes the choice of systems programmers?

SB: The point of all programming language safety features, be it Java-like automatic memory management or Rust's type system that enforces a discipline on pointers, is to avoid unintended state and, as a result of that state, unintended execution from that state onward. The difference between the languages and approaches is what kind of unintended state is being prevented and how this is done.

Historically, it was very easy for a programmer to unwittingly create unintended state. Classic ISAs use contents of memory or registers as addresses to access memory "randomly," i.e., in arbitrary order and without checking what, if anything, was previously stored in that memory and when or how it got there. C/C++ exposed this indirect memory addressing through pointers, which could point practically anywhere and allowed nearly arbitrary arithmetic to be applied to them. Reasoning about code—for example, what the code would do on all inputs hitting a module's boundary—in the presence of arbitrary pointers is very hard (see Hind's 2001 survey [15]). The power of arbitrary indirect memory references is so great that it's possible to (re)compile any program into just x86 MOV instructions and a single JMP or an equivalent way of looping backwards [16], which is, of course, really bad news for program analysis.

Java approached this problem by abstracting away almost all indirect memory references, to heavily restrict what memory addresses the CPU might access on behalf of the program (notionally mediated by the JVM, but also observed by JIT-ed code). To do so, it took memory management away from the programmer, which made it less desirable for OS programming,

where managing memory is a significant part of the task, and a single automated way of doing it just does not fit all needs. Rust, via its type system, controls pointers in a different way, but for the same purpose: restrict where and when indirect memory references can point so that they become tractable, unlike C's pointers or assembly's indirect MOVs [16].

In each case, the language makes memory-corrupting references hard or impossible for the programmer to create in ordinary code. However, as we've seen with web programming, memory safety alone does not preclude abuse of complex interfaces, and can actually make exploiting these interfaces easier, because the attacker doesn't need to worry about crashing the system with a poorly crafted input. We often forget that memory safety without a clear understanding of what inputs and outputs are legal works both ways and can easily favor the attacker.

There is definitely a LangSec perspective on this: IPCs are data languages, and whatever Rust or any other compiler can do is all done for the purpose of consuming these languages safely and not letting them drive unintended computation in a module or microservice.

So the question is, once again: regardless of whatever kinds of checks can be done, what constitutes expected and valid IPC messages that, once validated, will cause only predictable system behaviors and no other "weird" behaviors? Can these expectations be precisely and unambiguously formulated and checked with tractable code, which could itself be checked for correctness?

Without a clear LangSec model of the inputs, validating IPC messages becomes an ill-defined game of guessing which kinds of memory corruption or command injection to mitigate, for example, by making the hardware explicitly protect some address ranges from access by all code except specially designated code parts (e.g., via x86 ring contexts). But what happens in other ranges and contexts? How can one guarantee that corruption spreading there would not trick a legitimately placed privileged ("ringed") deputy into corrupting the protected region by passing it some unexpected inputs? This is a really hard question to answer, and it needs higher-level models of intended input-driven behaviors.

So compilers and build environments in general should absolutely be doing more work to make sure only intended state occurs, and it's a great thing that they do.

LangSec, for its part, helps formulate what is and can be the intended, tractably checkable state when dealing with inputs, and helps system, protocol, and application designers avoid situations where ensuring predictability of input-handling code becomes unsolvable. So LangSec has a lot of work to do and many programming fields to help secure.

References

[1] The Sixth Workshop on Language-Theoretic Security (LangSec), at IEEE Security & Privacy (May 2020): http://spw20.langsec.org/workshop-program.html.

[2] J. Chen, V. Paxson, J. Jiang, "Composition Kills: A Case Study of Email Send Authentication," 29th USENIX Security Symposium (Security '20):https://www.usenix.org/conference/usenixsecurity20/presentation/chen-jianjun.

[3] HTTP Desync attacks: https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[4] F5 vulnerability: https://research.nccgroup.com/2020/07/12/understanding-the-root-cause-of-f5-networks-k52145254-tmui-rce-vulnerability-cve-2020-5902/.

[5] "Psychic Paper" vulnerability: https://siguza.github.io/psychicpaper/.

[6] GitLab vulnerability: https://about.gitlab.com/blog/2020/03/30/how-to-exploit-parser-differentials/.

[7] Android Master Key vulnerabilities: http://www.saurik.com/id/17, http://www.saurik.com/id/18, and http://www.saurik.com/id/19.

[8] S. McQuistin, V. Band, D. Jacob, and C. Perkins, "Parsing Protocol Standards to Parse Standard Protocols," in *Proceedings of the Applied Networking Research Workshop (ANRW '20)*, pp. 25–31.

[9] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive Encoders for Context-Free Grammars," in *2015 IEEE CS Security and Privacy Workshops*: pp. 134–141: http://spw15.langsec.org/papers.html#unparse.

[10] N. Silvanovich and R. Farrow, "Interview with Natalie Silvanovich," *;login:*, vol. 43. no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/farrow-0.

[11] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security Applications of Formal Language Theory," *IEEE Systems Journal,* vol. 7, no. 3 (September 2013), pp. 489–500.

[12] F. B. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1 (February 2000), pp. 30–50: https://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf.

[13] S. Yegge, "Stevey's Google Platforms Rant," October 2011: https://gist.github.com/chitchcock/1281611.

[14] S. Maguire, "Protobuffers Are Wrong," Reasonably Polymorphic blog, October 10, 2018: https://reasonablypolymorphic.com/blog/protos-are-wrong/index.html.

[15] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, pp. 54–61: https://courses.cs.washington.edu/courses/cse501/15sp/papers/hind.pdf.

[16] C. Domas, MOVfuscator: https://github.com/xoreaxeaxeax/movfuscator.

[17] D. Kaminsky, M.L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks Against the Global X.509 Onfrastructure," in *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC 2010),* pp. 289–303: https://www.esat.kuleuven.be/cosic/publications/article-1432.pdf.