Research Statement

Paul Gazzillo

Fall 2017

1 Introduction

My goal is to make it easier to create safe and secure programs. As computing technology becomes further integrated with our daily lives, we are subjected to increasingly severe security and privacy risks. The recent Equifax breach put 143 million people at risk, and it was due to a vulnerability that lived in source code for nearly a decade [8]. Minor concurrency errors by developers using the Ethereum cryptocurrency cost users millions of dollars in the Parity Wallet attacks [18]. Internet-enabled devices have made cars, medical devices, and even guns susceptible to potentially life-threatening takeovers by remote hackers [9, 10]. Security guru Bruce Schneier puts it eloquently: "As everything turns into a computer, computer security becomes everything security" [20].

How can we protect software systems from these risks? Auditing software manually is expensive and must to be repeated for new versions. Antivirus software and intrusion detection systems do not stop as-yet-unknown vulnerabilities. A complete solution requires developing far more secure software from the start. To do so, developers need automated software tools for security analysis, bug finding, verification, and more. Automating these tasks makes development of secure software easier, cheaper, and less error-prone. Moreover, automated tools can be applied to legacy software built without a focus on security.

My research solves problems in security, systems, and software engineering with techniques from programming languages and compilers. I have co-developed an analysis that automates finding side channels [2]. This enables automated security auditing for leaks of secret data such as passwords and encryption keys. I have co-developed techniques that add concurrency safely and automatically to *smart contract* programs that run on cryptocurrency blockchains [4, 5]. I have solved the decades-old problem of parsing both C proper and the C preprocessor [7], lauded as a milestone in its recognition as a SIGPLAN Research Highlight [17]. Along with my research on build-system analysis [6], my parsing work provides the foundation of automated software tools for critical, highly-configurable software systems such as the Linux kernel and the Apache web server.

I intend to further advance these lines of research by developing practical static analyses for safe, secure software as well as tools for highly-configurable systems. I also intend to apply insights from this research to language design. Much of the challenge in making these tools practical are language features that inhibit automated software tools, features like reflection in Java, the C preprocessor, and pointer arithmetic in C/C++. Such features are useful to developers but easily abused. By making languages easier to analyze, tools will be more effective and developers will find it easier to reason about programs.

2 Research Projects

Detecting side-channel attacks. A *side channel* is an observable aspect of a program's execution, distinct from its input or output. Running time, memory usage, power consumption, network packet count, and more can all be side channels. If one of these happens to be correlated with secret information processed by the program, an attacker can exploit this to discover the secret, e.g., an encryption key or a password. For example, researchers infected a brand of smart light bulbs with a worm after discovering the firmware update key via a side channel [15]. Such side-channel attacks are particularly insidious. *Even when a program is functionally correct*, the performance characteristics of its algorithm may be a security vulnerability.

I, along with my colleagues, introduced a formal framework, algorithm, and implementation called Blazer for proving a program safe from timing channel attacks [2]. Programs have complex running time behavior that depends on their input, making it difficult to reason about running time for all inputs. Blazer works by decomposing the program into partitions according to groups of public inputs. It then proves the safety of each partition separately. A formal framework guarantees that proving each partition safe is enough to conclude the entire program is safe, given the right kind of partitioning function. When it cannot prove safety, Blazer switches to proving the existence of a side channel by partitioning on groups of secret inputs. In this case, finding differences in running time bounds between two partitions is sufficient for identifying a potential attack. This work is important for giving developers automated tools to perform security audits on their programs.

Adding concurrency to smart contracts. Smart contracts automate the execution or negotiation of a contract and have become a popular addition to blockchains. The finance industry is investigating them to automate financial transactions [14], and some banks have started using them [19]. A smart contract can be seen as a concurrent shared object stored indefinitely on a blockchain. But in the most popular smart contract platform, Ethereum, no concurrency control is built into the language. To make consensus easy, all operations are performed *sequentially*, a disaster for throughput.

Concurrency is notoriously hard to get right, so it was probably wise to support sequential reasoning in smart contracts. Work by me and my colleagues improves performance without complicating reasoning by adding concurrency to smart contracts *automatically* [4, 5], adapting techniques from software transactional memory. Data structures used by the smart contract are invisibly replaced with concurrent implementations, requiring no modification of the original smart contract program. Operations are then executed with speculative concurrency to exploit parallelism, producing a safe, serializable schedule. This schedule is published to the blockchain itself enabling all participants in the blockchain to benefit from improved concurrency. This work makes concurrent executions of smart contracts safe and improves throughput when contention is moderate, completely invisibly to the programmer.

Analysis infrastructure for highly-configurable C systems. Highly-configurable systems, e.g., the Linux kernel, form our most critical infrastructure, underpinning everything from high-performance computing clusters to Internet-of-things devices. Keeping these systems secure and reliable with automated tools is essential. However, their high degree of configurability leaves most critical software without comprehensive tool support. The problem is that most software tools do not scale to the colossal number of configurations of large systems. Recent work has shown that bugs, including null pointer errors and buffer overflows, can and do appear in only specific configurations [1]. With millions of configurations in complex systems like Linux, there are simply too many to analyze individually. Instead, my goal is to make tools that work on all configurations *simultaneously.*

The C preprocessor is used extensively to implement configurable code. Large systems like Linux use thousands of preprocessor macros and conditional compilation directives scattered throughout each C file. But parsing C and the preprocessor language together had remained a long-standing problem. To solve this problem, I developed new preprocessing and parsing algorithms and built an efficient, full-featured parser generator called SuperC [7]. SuperC is the first complete solution to parsing both C proper and the preprocessor and, despite its name, it can be used to generate parsers for other languages as well. The next challenge was tackling Makefile configurability. Large systems like Linux have thousands of C files, which are built by Makefiles according to complex relationships between thousands of configuration options. I developed and implemented a new static analysis for Makefiles [6] that is the first to find the complete set of configurations from Kbuild, the build system used by Linux, BusyBox, coreboot, and others. Together, SuperC and Kmax form the foundation on which to build bug finders, security analyses, translation tools, and more. These tools are essential for automating software analysis on critical C systems such as the Linux kernel and the Apache web server.

3 Research Agenda

My goal over the next five years is to evolve the way programmers write high-quality software. I plan to achieve this through practical applications of program analysis to software development and by advancing language design.

3.1 Practical Program Analysis

Automated security analysis. Security audits of software are important, and making them automated is necessary to lower the economic barriers to conducting them. I plan to continue developing analyses to discover side-channel attacks not just during development, but also during the testing phase using automated fuzz testing. The benefit of this approach is that it can be used to find vulnerabilities when a static code review is too imprecise or yields too many false alarms to be reliable. Dynamic analysis promises more precision; it can find concrete examples of inputs that demonstrate a vulnerability. The tradeoff is that fuzz testing can take hours or days and still miss the side channel. Given the limitations of current static approaches on large, real-world programs that use complex language features, such dynamic analysis is necessary, at least when static analyses fail. I also intend to further develop code review tools, such as information flow, to make them more practical for realistic usage by non-experts. Additionally, software engineering paradigms need to integrate such tools in the development process to make them useable without interfering with productivity. I plan to design such processes as to maximize the benefit of automated security auditing for large-scale software. Together, this work will help protect our software infrastructure from exploitation by malicious actors.

Program analysis for highly-configurable software. My work on SuperC and Kmax pave the way for static analysis tools for highly-configurable software. I plan to build on this foundation to develop bug finding and security analysis tools for such software. My goal is first to build an analysis framework that supports configurability and adapt existing static analyses to work on this framework. The challenge is that configurations affect all parts of the program, including types, function definitions, statements, etc, necessitating careful modifications to analysis algorithms. Worse, supporting all configurations introduces a new source of state explosion, requiring new tradeoffs in scalability and precision. These challenges are supported by efficient language representations and *configuration-sensitive* analyses that manage the configuration explosion problem. The next step is to use this framework to build tools that translate from error-prone, manual implementations of configurability to new languages for expressing complex system configuration. To achieve these goals, I have submitted a collaborative grant proposal to build the front-end infrastructure, bug finding tools, and translation tools. These efforts will yield better software tools for critical systems such as the Linux kernel, the Apache web server, and other complex systems.

3.2 Language Design

Language Extensions for Configurable Software. Using the preprocessor for implementing configurable systems in C makes for brittle, error-prone code, and I want to assist developers in moving away from preprocessor usage. Previous work has made attempts to either eliminate preprocessor usage or replace it with a new preprocessor [12], while work on variational programming explores new constructs for expressing configurability [3]. Inspired by these efforts, I propose language extensions that will replace preprocessor usage and let developers implement configurations directly in C. Compilers will then be able to check such usage and generate compiler-optimized code for these constructs, freeing the developer from manual and error-prone implementation using the preprocessor. One of the main obstacles to adoption of language extensions is the millions of lines of legacy code. The analysis infrastructure described above will provide the necessary tools to perform translation of much preprocessor usage automatically.

Safer smart contracts. While my previous work invisibly adds concurrency to individual contract operations, this does not protect against contracts that require atomicity across operations. Several smart contracts have been exploited, because their programmers made incorrect assumptions about atomicity [16]. My objective is to enable developers to provide verifiable specifications of behavior by building on existing smart contract verification work [11]. These specifications are to be published to the blockchain and automatically checked by participants in the network. What makes this particularly challenging is the dynamic nature of operations on smart contracts. Because smart contracts are immutable, developers use what is analogous to dynamic linking: contracts make a dynamic call to a subcontract referenced by a pointer. This enables dynamic updating of contracts that may violate the invariants of the original subcontract, exposing users who agreed to the original contract to an unsafe subcontract. Integrating contract specifications enables such dynamic linking to be checked whenever a subcontract is updated, automatically protecting its users. Blockchain smart contracts are essentially a platform for distributed programming. I would like to expore safe smart contracts for reliable distributed systems, along the lines of other concurrency paradigms in programming languages such as Go, Erlang, and Cilk.

Designing analyzable programming languages. Programming languages have features that lead to inferior performance of software tools. For sound static analyses, features like pointer arithmetic in C or reflection in Java often force tools to be too imprecise or resort to *soundiness*, i.e., soundness for only a subset of the program's executions. New designs can help minimize the downsides of certain program behavior; e.g., Java's memory safety eliminates the need to analyze pointer arithmetic. But even Java has features, e.g., reflection, that confound static tools. My goal is to design programming languages that are more amenable to static analysis. This will facilitate automated software engineering tasks, like bug finding and refactoring, and lead to safer programming. The key challenge is to preserve programmer freedom and expressivity and find the intersection of language features that are easy for both humans and tools to reason about.

Designing a language design from scratch is expensive and risky. Rather than develop a new language from scratch, I plan to work from existing languages. By defining an easier-to-analyze subset of a language such as C, my research will first explore translation from the full language to this analyzable subset, providing support for legacy code. Then, by building on this language subset, I will develop new language extensions that are transformed into the subset, rather than creating a new compiler from scratch. I recognize that new language adoption depends on far more than any technical features: real-world language choices depend on consideration for legacy code, library availability, community support, and many other social and cognitive factors. I plan to use sociological studies, for instance from the SocioPLT project [13], to help guide design decisions. By gradually making existing languages easier for analysis tools, I hope to make automated security auditing, verification, and bug finding well-integrated into software development.

4 Conclusion

My overall research agenda is to give developers the right tools and languages to make safe and secure software. I am optimistic about the prospect of software tools to improve the security and reliability of software. While the old adage says, “a poor workman blames his tools,” I believe good tools make great programmers even better. But I recognize that tools alone are not enough. Programming language design requires human studies to understand how programmers produce quality code efficiently. Sensitive software systems require good design and secure engineering practices. Most importantly, education is the cornerstone of quality software. Developers need training in fundamental computer science, critical thinking skills, creativity, and good engineering practices. Combined with the right languages and tools, the next generation of developers will make the most reliable and secure software possible.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE ’14, pages 421–432, New York, NY, USA, 2014. ACM.

[2] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 362–375, New York, NY, USA, 2017. ACM.

[3] Sheng Chen, Martin Erwig, and Eric Walkingshaw. A calculus for variational programming. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 6:1–6:28, 2016.

[4] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC ’17, pages 303–312, New York, NY, USA, 2017. ACM.

[5] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Brief announcement: Proust: A design space for highly-concurrent transactional data structures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC ’17, pages 251–253, New York, NY, USA, 2017. ACM.

[6] Paul Gazzillo. Kmax: Finding all configurations of kbuild makefiles statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 279–290, New York, NY, USA, 2017. ACM.

[7] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 323–334, New York, NY, USA, 2012. ACM.

[8] René Gielen. Apache Struts Statement on Equifax Security Breach. `https://blogs.apache.org/foundation/entry/apache-struts-statement-on-equifax`, 2017. [Online; accessed 05-Nov-2017].

[9] Andy Greenberg. Hackers Can Disable a Sniper Rifle–Or Change Its Target. `https://www.wired.com/2015/07/hackers-can-disable-sniper-rifleor-change-target/`, 2015. [Online; accessed 05-Nov-2017].

[10] Andy Greenberg. Hackers Remotely Kill a Jeep on the Highway–with Me In It. `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, 2015. [Online; accessed 05-Nov-2017].

[11] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[12] Bill McCloskey and Eric Brewer. ASTEC: A new approach to refactoring C. pages 21–30, September 2005.

[13] Leo Meyerovich and Ari Rabkin. SocioPLT. `https://lmeyerov.github.io/projects/socioplt/viz/index.html`. [Online; accessed 27-Nov-2017].

[14] Giulio Prisco. Smart Contracts and the Future of Banking. `http://www.nasdaq.com/article/smart-contracts-and-the-future-of-banking-cm849118`, 2017. [Online; accessed 27-Nov-2017].

[15] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. `http://iotworm.eyalro.net/`, 2016. [Online; accessed 05-Nov-2017].

[16] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. 02 2017.

[17] SIGPLAN Research Highlights Papers. `http://www.sigplan.org/Highlights/Papers/`, 2012. [Online; accessed 05-Nov-2017].

[18] Emin Gün Sirer. Parity's Wallet Bug is not Alone. `http://hackingdistributed.com/2017/07/20/parity-wallet-not-alone/`, 2017. [Online; accessed 05-Nov-2017].

[19] Sujha Sundararajan. State Bank of India to Beta Test Blockchain Smart Contracts Next Month. `https://www.coindesk.com/state-bank-of-india-to-roll-out-smart-contracts-and-blockchain-kyc/`, 2017. [Online; accessed 27-Nov-2017].

[20] Gabriela Vatu. Bruce Schneier Says Government Involvement in Coding Is Coming. `https://www.schneier.com/news/archives/2017/02/bruce_schneier_says_.html`, 2017. [Online; accessed 05-Nov-2017].