William Harris — Research Statement

Modern programs are complex and, as a result, often untrustworthy (i.e. incorrect or insecure). In principle, modern research in programming languages provides powerful techniques for verifying and synthesizing correct programs, but in practice, such techniques have subtle limitations. **My research extends and applies such techniques to enable programmers to build practical, trustworthy programs.** My approach is motivated by the practical needs for trustworthy programs that arise from computer security and software engineering: I have collaborated with researchers in systems security [3, 4, 5, 7, 10] and operating systems [6, 8, 9], and have studied the needs of computer end-users [1, 2], to identify the critical programming problems faced by developers. To address each programming problem, I have developed (1) a novel programming language that enables a programmer to describe the correctness and security properties of their program simply and (2) techniques that automatically translate properties described by a programmer in the language to an executable program that satisfies the properties.

I will now describe in more detail how I have applied my approach to accomplish three goals: (1) enabling programmers of security-critical applications to write secure programs, (2) enabling end-users to write programs that transform tabular data, and (3) enabling programmers of performance-critical applications to validate program optimizations.

Trustworthy programs from untrusted components Programs storing sensitive information are constantly the targets of attacks that exploit vulnerabilities in complex, untrustworthy program modules (i.e., classes or functions). The systems-security community has demonstrated that rather than *removing* all vulnerabilities in a program, a program can be *instrumented* to use powerful programming *primitives* provided by new *security-conscious operating systems* to preserve the security of the program's information. However, such systems require the programmer to determine if a program instrumented to use the system's primitives satisfies the programmer's high-level notion of a security guarantee (i.e. *policy*), which the programmer must leave as informal and implicit; the gap between instrumented programs and high-level policies has led even the developers of such systems to instrument programs with behaviors that later surprised them.

I have developed programming tools for two types of security-conscious operating systems, namely *capability* and *decentralized-information-flow* systems. My tools significantly ease the task of programming for such systems; they enable programmers to explicitly declare the security policies that their program must satisfy separately from the program itself, and automatically obtain a program that satisfies the policies. Such tools greatly ease the task of determining what attacks an attacker can or cannot carry out against a program, and will hopefully encourage the adoption of security-conscious systems in the future.

The Capsicum capability system (now included in the FreeBSD operating system) provides primitives based on *capabilities* (i.e., an object paired with a restricted set of operations that may be performed on it). A program can use capabilities to limit the operations that untrusted program modules can perform on sensitive information (e.g., a program can use capabilities to ensure that an untrusted compression function can only write sensitive information to a user's terminal but not to a network connection). However, a programmer who instruments their program to execute on Capsicum must manually partition their program into modules that execute with different capabilities. The programmer must then informally determine if their partitioned program will only perform sequences of operations on sensitive information that constitute secure behavior. To address this programming problem, I designed (1) a policy language that allows a programmer to explicitly describe sequences of operations on sensitive information that a program should or should not be able to carry out, and (2) a *program instrumenter* that takes an uninstrumented program and a policy, and automatically instruments the program to satisfy the policy [5]. The key challenge in designing such an instrumenter was to design an instrumentation algorithm that, from only a *declarative* policy, generates *executable* code that (1) maintains relevant information about the current capabilities of modules and (2) uses the information to update capabilities as necessary to satisfy a policy. I worked with the Capsicum developers to write a set of policies for heavily used UNIX utilities; each policy forbids vulnerabilities in recent versions of the utilities. I worked with a security-evaluation team at MIT Lincoln Laboratory (MITLL) to write a policy for the PHP interpreter as part of an experimental secure system

developed by MITLL. I applied my instrumenter to each program and its policy to produce an instrumented program that satisfies its policy.

Decentralized Information Flow Control (DIFC) systems provide primitives that a program can use to preserve the *secrecy* and *integrity* of its sensitive information (i.e., the program can ensure that its sensitive information is not leaked or corrupted). A program calls DIFC primitives to direct how the operating system maintains a *partially-ordered lattice structure of labels*, which the operating system consults to determine if information can flow between system objects. Manually translating a secrecy or integrity policy to a program that correctly maintains such a lattice is a difficult programming problem. To address this problem, I developed (1) a policy language for the Flume DIFC operating system that allows a programmer to declaratively describe desired secrecy and integrity policies for their program's information, and (2) a program instrumenter that takes an uninstrumented program and a policy, and automatically instruments a program that directs the operating system to maintain labels so that the policy is satisfied [3]. The key challenge in designing such an instrumenter was to design an instrumentation algorithm that could reason about the powerful primitives provided by DIFC systems, which, unlike capabilities, are based on *partial orderings and transitive relations over an unbounded set of objects*. Using the policy language and instrumenter, I instrumented label-manipulating "launchers" for the ClamAV virus scanner and the OpenVPN network client, and the Apache multi-process module.

Capsicum and Flume (along with other DIFC and *tagged memory* systems) demonstrate that the systems-security community continues to develop powerful secure operating systems, each with distinct security primitives. Rather than requiring a developer of a new operating system to develop a policy language and instrumenter for their system from scratch, I designed an *instrumenter-generator* that a developer can apply to automatically obtain a policy language and instrumenter for their operating system [4]. A developer provides to the generator (1) the representation of their system's security-relevant state (e.g., a DIFC developer would provide the representation of a label), (2) the set of primitives that the system provides to an application, and (3) a declarative description of how each primitive updates the security-relevant state of the system. The generator provides (1) a policy language defined over features of the state provided by the developer and (2) a program instrumenter, which operates as described above. They key challenge in designing the generator was to find a language for describing system primitives that was expressive enough to describe apparently disparate security-conscious systems, along with a "meta-policy language" and "meta-instrumentation algorithm" that could be instantiated to a policy language and instrumenter for any system that a developer could describe; we used an extension of first-order logic and a *game-based synthesis* algorithm. We applied our instrumenter-generator to automatically generate an instrumenter for the HiStar DIFC system from a declarative specification of HiStar primitives that is less than 1,500 lines.

Programs from user-provided examples Many computer users need to perform repetitive tasks, but are not willing, and should not be expected, to dedicate the considerable time and resources required to master a conventional programming language. As a Research Intern at Microsoft Research, I designed (1) a domain-specific language capable of expressing popular transformations in the *layout of spreadsheet tables* and (2) a program synthesizer that takes a set of example inputs and output tables from a user, and provides to the user a general program that implements the examples [2]. The key challenge was to define a language whose programs could be inferred efficiently from only examples, but could also describe practical spreadsheet transformations, which typically treat a spreadsheet as partially-structured data (i.e., most end-users do not use a spreadsheet table as a relational table). I applied the program synthesizer to automatically synthesize programs that solved over 50 requests that had been posted to popular Excel help forums.

The language and synthesizer for transforming table layouts is part of a collective effort at Microsoft Research to synthesize from examples programs that perform a variety of everyday tasks required by end-users, such as matching and replacing rich patterns in strings and using knowledge of popular semantic domains (e.g., calendar dates) [1]. I drove the design and development of the layout synthesizer, which was described in a paper [2] that was one of two papers recognized collectively as a CACM Research Highlight.

Explaining program optimizations Developers of mature, performance-critical applications spend considerable time and resources optimizing the performance of their application, but struggle to determine that their optimization preserves the functional behavior of their program. Previous work by systems researchers at Wisconsin found that developers often significantly optimize the performance of their program by changing only how their program calls library functions, whose implementations either are not available, or are too complex to be analyzed. We used this insight to develop an *interactive optimization validator* that takes an original and optimized program and presents a condition on only the library functions called by the programs such that if the condition holds, then the original and optimized programs are equivalent [6]. A programmer using the tool can accept the condition as a formal assumption for why they believe their optimization to be valid, or can refute the assumption and require the validator to find a new condition that is sufficient to support the optimization. The key challenge was to design a validator that can suggest simple conditions over purely the library functions called in an optimization based on how such functions are *used*, not how the functions are *implemented*. We applied our validator to validate optimizations submitted in bug reports for the Apache web server, Mozilla software suite, and MySQL database.

Classical program verification In addition to developing new languages for specifying program properties, I have proposed and developed automatic methods for checking that a program satisfies conventional, fundamental properties, including safety properties [9, 10] (e.g., memory safety) and termination on all inputs [8]. I have omitted detailed descriptions of this work, for brevity.

Future work Based on my previous work, I believe that the following strategy is effective for building practical, trustworthy programs: (1) identify programs that require rigorous guarantees of correctness; (2) decompose the problem of verifying that such a program into multiple *verification sub-problems* that exhibit a tradeoff in scalability vs. tractability, and develop solutions to each problem; (3) compose the guarantees obtained by solving each sub-problem to verify the desired guarantee of the entire program. My current work implements the intermediate steps of this strategy in the domains of software security and interactive verification. I plan to pursue the following research goals as next steps:

Fine-grained, system-wide information-flow For a program to completely protect the secrecy of its sensitive information, the program must ensure that when it releases sensitive information in a data structure to other objects on its system, the sensitive information is stored only in allowed objects on the system. E.g., a program storing multiple users' credit card information may need to ensure that if it releases the information stored for a particular user in a table, then the information should only be stored in the user's home directory. DIFC *languages* (e.g., *Jif*) provide powerful techniques for reasoning about how information flows between data structures, but cannot reason about information released by a program to other system objects. DIFC *operating systems* (e.g., HiStar) provide primitives for controlling how information flows between system objects, but cannot control how information flows between program data structures. A framework that composes a DIFC language and DIFC system, using partly a program instrumenter developed in my previous work in programming for DIFC systems, could allow a programmer to write programs with previously unattainable information-flow guarantees that connect the information in program data structures to the information in objects throughout a system. Designing such a framework will require collaboration between researchers in programming languages, interactive and automatic verification, and operating systems.

Selectively-interactive verification Verifying desired properties of a program requires both (1) efficiently summarizing the effects of large segments of the simple code (e.g., instructions that execute arithmetic operations in sequence) and (2) inferring subtle properties of relatively small segments of complex code (e.g., code that conditionally calls a recursive function). In practice, automatic-verification algorithms summarize simple code well, but struggle to infer required subtle properties of complex code. By decomposing a verification problem into the two subproblems described above, selectively asking a programmer for assistance in finding subtle properties, and composing the programmer's response with summaries inferred by the analysis automatically, one may be able to verify non-trivial properties of a significant class of practical programs that are beyond the scope of automatic algorithms, but with

significantly less effort than by proving correctness of the entire program manually. Designing practical interactive verification algorithms will require collaboration between experts in program verification, software engineering, and human-computer interaction.

Hardware accelerators from optimization rules The hardware community has significantly optimized program performance by designing architectures with *hardware accelerators*, i.e., specialized circuits that a program can execute to perform the effect of a sequence of instructions more efficiently than executing the explicit sequence. However, an accelerator designer faces significant challenges: accelerators often aggressively optimize the sequence of instructions executed by a program, but it is often unclear if an accelerator preserves the semantics of all sequences of instructions that a program may execute. Furthermore, an accelerator designer must choose from a enormous space of circuits to add to an architecture, and must choose under what conditions each circuit should execute.

Accelerator design could be significantly improved if an accelerator designer could specify the operations of an accelerator as a set of *declarative optimization rules*, and automatically obtain: (1) a proof that optimizations allowed by the rules preserve program behavior, (2) a predictive model that the designer can use to tune accelerator behavior on particular benchmarks, and (3) an executable accelerator that implements the rules. A framework that synthesizes a behavior-preserving accelerator from optimization rules will need to use distinct verification techniques for reasoning about the correctness of optimization rules and for synthesizing accelerator circuits from optimization rules, and compose the guarantees of these techniques to synthesize a correct accelerator circuit. Designing such a framework will require collaboration between experts in verification, synthesis, and systems architecture.

References

[1] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[2] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Programming Language Design and Implementation – PLDI*, pages 317–328. ACM, 2011.

[3] W. R. Harris, S. Jha, and T. W. Reps. DIFC programs by automatic instrumentation. In *Computer and Communications Security – CCS*, pages 284–296. ACM, 2010.

[4] W. R. Harris, S. Jha, and T. W. Reps. Secure programming via visibly pushdown safety games. In *Computer Aided Verification – CAV*, pages 581–598. Springer, 2012.

[5] W. R. Harris, S. Jha, T. W. Reps, J. Anderson, and R. N. M. Watson. Declarative, temporal, and practical programming with capabilities. In *IEEE Symposium on Security and Privacy*, pages 18–32. IEEE Computer Society, 2013.

[6] W. R. Harris, G. Jin, S. Lu, and S. Jha. Validating library usage interactively. In *Computer Aided Verification – CAV*, pages 796–812. Springer, 2013.

[7] W. R. Harris, N. Kidd, S. Chaki, S. Jha, and T. W. Reps. Verifying information flow control over unbounded processes. In *Formal Methods – FM*, pages 773–789. Springer, 2009.

[8] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *Static Analysis Symposium – SAS*, pages 304–319. Springer, 2010.

[9] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *Principles of Programming Languages – POPL*, pages 71–82. ACM, 2010.

[10] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty. Security challenges in automotive hardware/software architecture design. In *DATE*, pages 458–463, 2013.