1 Introduction

In the high profile 2017 *Equifax* attack, millions of individuals' private data was stolen, costing the firm hundreds of millions of dollars in remediation efforts. This attack leveraged a *code injection* exploit in Apache Struts (CVE-2017-5638), and is just one of over 8,200 similar code injection exploits discovered in recent years in popular software [74]. Code injection vulnerabilities have been exploited in repeated attacks on US election systems [33,54,71,95], in the theft of sensitive financial data [88], and in the theft of millions of credit card numbers [66]. In the past several years, code injection attacks have persistently been ranked at the top of the OWASP top ten most dangerous web flaws [75]. Injection attacks can be very damaging even for applications that are *not* traditionally considered a critical/important target (such as personal websites), because attackers can use them as footholds to launch more complicated attacks.

In a code injection attack, an adversary crafts a malicious input that gets interpreted by the application as code (rather than data). These weaknesses ("injection flaws") are so difficult to detect that rather than suggesting testing as a defense, OWASP instead suggests that developers try to avoid using APIs that might be targeted by attackers, or enforce site-wide input filtering. Consider again the Equifax hack: the underlying weakness that was exploited was originally introduced in 2011 and sat undetected in production around the world (not just at Equifax) for *six years* [7, 72]. While some experts blame Equifax for the (successful) attack — a patch had been released two months prior to the attack, but was not applied — one really has to ask: how is it possible that those critical vulnerabilities go unnoticed in production software for so long?

With the exception of safety-critical and similar "high assurance" software, general best practices call for developers to extensively test their applications, to perform code reviews, and perhaps to run static analyzers to detect potentially weak parts of their software. Unfortunately, testing is a never-ending process: how do developers know that they've truly tested all input scenarios? To catch code injection exploits just-in-time, researchers have proposed deploying dynamic *taint tracking* frameworks, which dynamically track information flow, ensuring that no untrusted input flows into sensitive parts of applications (e.g. interpreters) [16, 57, 70, 84, 91, 99]. However, these approaches are generally not adopted due to prohibitive runtime overhead: even the most performant can impose a slowdown of at least 10-20% and often far more [16, 36, 45, 64]. While some static analysis tools have seen recent developer adoption (e.g. ErrorProne [55], FindBugs [102], and Coverity [28]), statically detecting code injection vulnerabilities without false positives is a very challenging problem since these tools must perform interprocedural dataflow analysis. For instance, static taint analysis [11, 94, 101] often suffers from false positives and/or false negatives.

I propose to create a new approach for augmenting existing tests, AMPLIFY: **A**utomatically **M**odifying **P**rogram testcases to reveа**L** **I**nformation **F**low vulerabilit**Y**(s), in order to help developers detect code injection vulnerabilities in their applications. My key insight is that *by combining dynamic taint tracking with existing, developer-written tests, I can enable developers to detect code injection vulnerabilities*. Combining dynamic analysis with existing test cases will allow me to avoid the fundamental limitations that have hindered prior work. Figure 1 shows an overview of this project. While prior work has used the term test *amplification* to refer to techniques that automatically inject exceptions or system callbacks [3, 106, 107], I use the term *amplification* far more broadly here. Given an existing application and its test suite as an input, AMPLIFY will increase the *depth* of these tests by automatically inferring new security-based oracles that will be checked by existing tests (**Thrust 1**). Combining these new oracles with existing test cases may or may not expose vulnerabilities, hence AMPLIFY will also increase the *breadth* of these tests by systematically broadening the inputs checked by each test case, mutating existing (non-defect revealing) test cases into new cases that might reveal vulnerabilities, and checking if those tests reveal true vulnerabilities (**Thrust 2**). AMPLIFY will be designed to integrate directly with test cases that developers already write and the existing, resource-limited environments that developers run and debug those tests in, both on their desktop computers and in continuous integration environments (**Thrust 3**). Finally, AMPLIFY will create low-overhead monitoring probes for production software that will serve as a last line of defense to harden

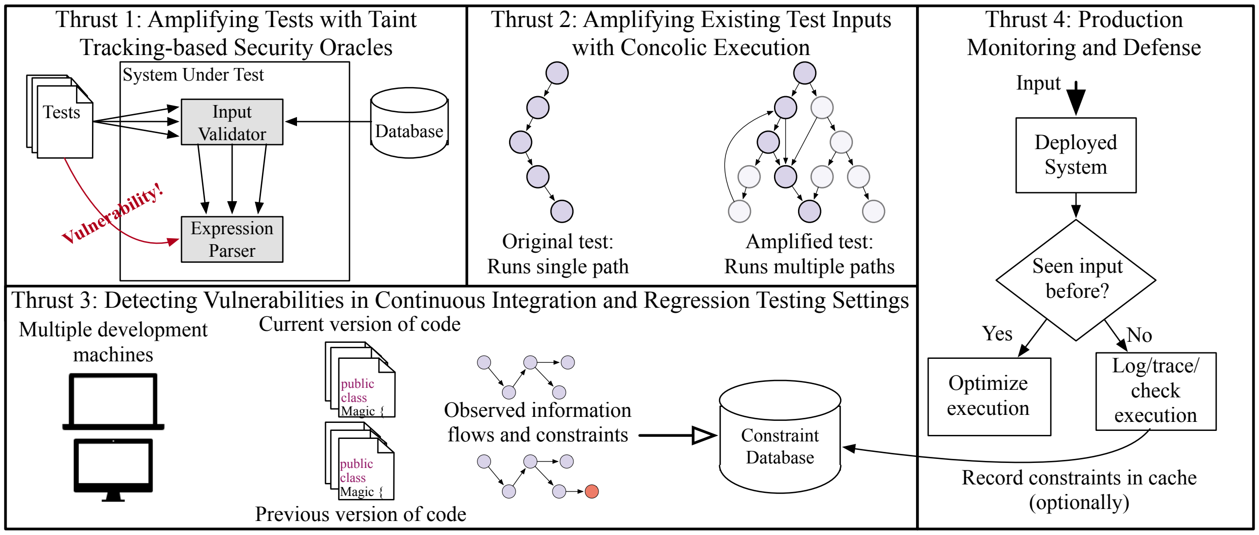
Thrust 1: Amplifying Tests with Taint Tracking-based Security Oracles
System Under Test
Tests
Input Validator
Database
Vulnerability!
Expression Parser
Thrust 2: Amplifying Existing Test Inputs with Concolic Execution
Original test: Runs single path
Amplified test: Runs multiple paths
Thrust 4: Production Monitoring and Defense
Input
Deployed System
Seen input before?
Yes
No
Optimize execution
Log/trace/ check execution
Record constraints in cache (optionally)
Thrust 3: Detecting Vulnerabilities in Continuous Integration and Regression Testing Settings
Multiple development machines
Current version of code
public class Magic {
Previous version of code
Observed information flows and constraints
Constraint Database

Figure 1: High level summary of the four proposed thrusts.

software against critical zero-day vulnerabilities that might have escaped testing (**Thrust 4**). Although my approaches will be language agnostic, the tools that I create will target applications running on the JVM (e.g. written in languages like Java or Scala) based on its popularity. This project will continue my strong tradition of releasing all of my tools as open source software [27].

The distinction between high-assurance and general purpose software grows thinner with every day as software is increasingly relied on at different levels of our cyber infrastructure. *My vision for testing and security research over the next decade is for efficient, automated tools that can be used by any developer to harden software against these complex forms of attacks.* This project builds on my prior research in improving the efficiency [17,19,22] and reliability [20] of existing test cases, laying a strong foundation for my career developing new approaches for improving the security of software through testing. Further, this project will support my ongoing mentoring and education activities that go far deeper than simply mentoring PhD students: within my university, I advise and mentor the largest CS-related undergraduate group [69]; outside of my university I have and will continue to organize undergraduate mentoring workshops that broaden the participation of underrepresented groups in computing research [25,82].

1.1 Intellectual Merit

The intellectual merit of this project includes: (1) A novel approach for performing dynamic dataflow analysis (taint tracking) in the context of developer-written test cases and the application of this approach for increasing the *depth* of existing tests with new oracles; (2) A new approach for tracking path constraints on variables in running applications and performing very lightweight concolic execution, generating new inputs that expand the *breadth* of existing tests; (3) An innovative methodology for integrating dynamic analysis tools in modern continuous integration and regression testing environments; and (4) A novel approach for creating lightweight production monitoring probes based on the behavior of existing tests.

This research agenda is driven by several key insights regarding the state-of-the-art of code injection vulnerability detection: (1) Platforms that were often ignored by hackers in the past (e.g. JVM) are increasingly targeted and hence worthy of tailored defenses; (2) The distinction between high-assurance and general-purpose software has blurred, and it is incredibly important that security testing can be performed by *all* developers, rather than only a limited few; (3) No purely automated test generation strategy will be able to expose all of the complex behavior that developer-written tests can, and instead, approaches should leverage both developer tests and automated test generation; (4) White box fuzzing (concolic execution) is often limited by inefficient fuzzer designs, greatly reducing the rate at which new program behaviors can be explored; (5) Security testing must, like other kinds of testing, be optimized for modern continuous integration environments, where developers get constant feedback on the quality of their code. Rather than focus on finding a handful of "million dollar bugs" in a small pool of high-assurance software, my goal is to find thousands of "thousand dollar bugs" in all kinds of software (and enable developers to do so themselves), finding as many high-impact bugs (like the Struts/Equifax vulnerability) as I can along the way.

The underlying advances in program analysis and software systems will create a platform for enabling others to pursue new research directions in testing, security and reliability. This project's core approach of applying taint tracking to test cases has the potential to influence how other dynamic analysis tools are used by developers. This project will involve graduate and undergraduate students and is expected to lead to at least two PhD theses. Each of the first two thrusts form natural dissertation topics that are sufficiently related to allow two PhD students to work collaboratively, yet sufficiently distinct to ensure that each of these two students will have clear thesis topics. Evaluation criteria are defined at the component and project level. My prior work demonstrates the feasibility of performing large scale studies of open source tests [17,19,20,60], and my pilot studies demonstrate the feasibility of reproducing real code injection vulnerabilities.

PI Qualifications. I have published 18 peer-reviewed papers on software testing and analysis, winning two Distinguished Paper awards (VMVM at ICSE 2014 [17] and HitoshiIO at ICPC 2016 [97]). I am an expert in building runtime systems for the JVM, as witnessed by my novel taint tracking system (PHOSPHOR, OOPSLA 2014 [16]), my unique taint tracking-based Android privacy system (PEBBLES, OSDI 2014 [92]) and by my checkpoint/rollback system (CROCHET, ECOOP 2018 [24]). For this project, my program analysis expertise is well complemented by my software testing background, including in flaky tests [17,19,20,48,60] and metamorphic testing [23,98]. I have a strong track record of disseminating tools (both open source and commercially) that are useful to and adopted by the community [27]. My research in regression test acceleration [17, 19] led to an industrial collaboration with ElectricCloud [44], and my recent research in flaky tests (DeFlaker, ICSE 2018 [20]) has sparked discussions with Google, Microsoft and Huawei. PHOSPHOR [14,16,18] has been adopted by many external researchers in the four years since its release, including in three theses [4,42,90] and several other publications [5,100]. Many researchers have already expressed interest in using CROCHET (published in July 2018), which is also freely available via GitHub [12].

2 Research Plan

To help understand how AMPLIFY will be used by developers, I provide a high-level overview of CVE-2017-5638 (the vulnerability used in the Equifax attack) [72]. Equifax's application was built using the popular open source Java-based Apache Struts framework, which among other things, manages templates that get rendered into web pages by its *OGNL* (Object Graph Notation Language) scripting engine. OGNL is very expressive, and allows these template expressions to call arbitrary Java functions. If an attacker can control a string that was parsed by the scripting engine, then the attacker can execute arbitrary Java code in the server process. Hence, it's very important that Struts does *not* allow untrusted inputs to flow into the OGNL engine (e.g. through input validation). In this case (CVE-2017-5638), there was a key flaw in Struts' error handling code, specifically, the code that handles invalid HTTP `Content-Type` headers, which passed the `Content-Type` header (which is part of the client's request) directly into the scripting engine to render a default error message. Since Struts did not escape that header (which would have prevented it from being interpreted), an attacker can execute arbitrary code (in that header) on the application server.

The state-of-the-art approach to mitigate code injection exploits like this one is to deploy the application with a dynamic *taint tracking* framework, which tracks the origin of values in memory. Developers add annotations that particular inputs should be tainted, and then the framework tracks the flow of those inputs through the application. If an input flows through a *sanitizer* (e.g. a method that checks for illegal inputs), then it might be deemed not a risk (and the taint tag dropped), because the sanitizer has ensured that it is not an attack. If a tainted input reaches a critical method in the program code (a "taint sink"), then the framework might throw an exception in order to prevent the untrusted input from reaching the critical code [57, 70, 84, 91, 99]. However, taint tracking is almost never deployed with production applications because: (1) Taint tracking imposes a non-trivial slowdown (of 3-50x [36, 39], although my optimized tool shows only 1-2x [16]); and (2) Taint tracking can raise false alarms if sanitizers are not annotated and even miss alarms if functions are incorrectly marked as sanitizers. Static taint analysis [11, 94, 101] often suffers from scalability issues and false positives, hindering its adoption by developers.

My goal is to allow developers to detect these vulnerabilities in their apps *before* releasing them, and without imposing an undue burden of false positives or excessive compute cycles. Consider how these developers might *normally* perform quality assurance: they write tests and perform code review. Although Struts had a moderate-sized test suite (with 46% statement coverage [6]), it did not reveal this vulnerability (before it was detected). My key insight is that I can use taint tracking *while running tests* to amplify the quality of those tests, detecting potentially vulnerable information flows, and then use input generation techniques to explore those flows and generate actual exploits demonstrating those vulnerabilities.

How will AMPLIFY help these developers? Consider: if the developers had written *any* test that made a request with an invalid `Content-Type` header, they would have exposed the behavior that allowed this untrusted input (the HTTP header) to flow into a sensitive sink (the OGNL engine). This test could be *amplified* adding new oracles (amplifying the *depth* of each test), specifically, asserting that no uncontrolled input could flow to a sensitive sink, which would then reveal the vulnerability (**Thrust 1**). In the event that there is no test with an invalid `Content-Type` header, but there is another test that makes any request at all against the Struts server (which is the case for Struts), AMPLIFY can amplify the *breadth* of the test, automatically mutating the existing test so that it makes its request with an invalid `Content-Type` header (**Thrust 2**). In either case, once the vulnerable information flow is detected,AMPLIFY can demonstrate that an attack is possible by re-executing the amplified test, trying to use specially crafted attack-strings in that information flow. This procedure will not require developers to annotate any input sanitizers, since AMPLIFY will instead repeatedly test the application with attack strings. AMPLIFY will fit directly into the continuous integration workflow used by the Struts developers, efficiently performing *regression* checking of these various properties while development occurs (**Thrust 3**). Based on the results of all of the testing performed pre-release, I will also consider how AMPLIFY can automatically generate efficient monitors to protect against (undetected) vulnerabilities in production.

2.1 Thrust 1: Testing for Code Injection Attacks with Dynamic Taint Tracking

Dynamic taint analysis is a powerful form of dynamic information flow analysis useful for identifying the origin of data during execution. Inputs to an application are "tainted," or labeled with a tag. These inputs can be determined in advance (*i.e.,* specifying that all values returned by some method are to be tainted), or completely dynamically (*i.e.,* using an arbitrary check at runtime to determine if a value should be tainted). As computations are performed, these labels are propagated through the system such that any new values derived from a tagged value also carry a tag derived from these source input tags. In this way, we can inspect any object and determine if it is derived from a tainted input by inspecting its label. Taint tracking has repeatedly been proposed as a defensive mechanism against code injection attacks [57,70,91,99]. In this context, taint tags are propagated from all user-controlled inputs, and if the framework observes a flow from an input to a sensitive function (e.g. one that parses and executes code), a violation is reported. However, dynamic taint tracking is not yet used in production runs because even the most performant dynamic taint tracking systems (like my own Phosphor [16]) still impose some performance overhead.

I propose to apply dynamic taint tracking to amplify the *depth* of existing (developer-written) test cases, adding new oracles to detect code injection vulnerable-code. In this model, developers do *not* need to write test cases that demonstrate an attack — instead, they need only write test cases that expose an information flow that is vulnerable to an attack. For instance, consider a recent Apache Struts vulnerability (CVE-2017-9791) that allowed user-provided web form input to flow directly into the OGNL engine. Struts includes a sample application for keeping track of the name of different people, this application can be used to demonstrate this vulnerability by placing an attack string in the "save person" form. To detect this vulnerability, we do *not* need to observe a test case that uses an attack string in the input, instead, we need only observe *any* test that saves *any* string through this form in order to observe the insecure information flow. Once this is detected, AMPLIFY can then re-execute and perturb the existing test case, substituting different values for the form field, attempting to demonstrate the exploit.

Although I will be able to build atop my existing taint tracking system, PHOSPHOR, there are a variety of challenges to successfully detecting injection attacks using taint tracking and existing test cases (Thrust 2 will also generate new inputs to amplify the breadth of existing tests). For instance: can we identify a comprehensive set of sources and sinks that will be generally applicable to different applications? Once I have run AMPLIFY on a variety of programs with a comprehensive set of sources and sinks, I expect that PHOSPHOR will report many violations, some of which may be benign — just because there is a data flow from a source to a sink, this may not be indicative of an attack vector. Hence, I will develop heuristics to help AMPLIFY filter and prioritize its reports. In some cases, flows might be trivially benign because the user input flows through a sanitizing function — I will develop new approaches to automatically detect likely sanitization functions, which can be treated as black boxes, or further scrutinized for safety. Taint tracking systems rely on an underlying set of propagation rules that define how taint tags are applied through data and/or control flow. There is no prior study of the effect of different propagation rules on the precision and recall of a high-level approach like AMPLIFY, and hence, I will experimentally evaluate different configurations to tune AMPLIFY. Finally, once the (suspected) attack vectors are detected, AMPLIFY will need to automatically re-execute the corresponding test(s) that exposed that flow and transform those benign test cases into attacks. At first, I will assume that each application runs in a single process (for the purposes of data flow tracking); 2.2.5 will expand AMPLIFY to consider distributed systems.

Preliminary Results. Much of the proposed work involves dynamic taint tracking, which will be build on my PHOSPHOR system [14,16,18]. PHOSPHOR is easy to deploy since it does not require specialized JVMs, and can be easily integrated with popular test execution frameworks like JUnit and TestNG [16]. PHOSPHOR is *complete*, in that it tracks taint tags through all variables and bytecode that runs in the JVM, in contrast to prior approaches that only tracked tags on Strings [37,56,57]. While PHOSPHOR does not track taint tags through native code (which is written in C/C++ and linked into the JVM binary), this limitation has not been a concern in practice as very few Java applications include native code. Real systems often receive inputs in a raw format (e.g. array of bytes), which then might be parsed into strings, then into character arrays, and then perhaps later back into strings, hence, it is important to track more than just strings. While previous systems for generic taint tracking have shown runtime slowdowns on the order 2-30x, PHOSPHOR's unique approach of bytecode instrumentation is amenable to aggressive JIT optimization, resulting in typical overheads on the order of only 10% (on DaCapo benchmarks; I have made significant optimizations to PHOSPHOR not represented in my original publication on it, which had reported an average overhead of 50% [16]). As a pilot study, I used PHOSPHOR to monitor Apache Struts while executing two recent and publicly disclosed attacks, CVE-2017-5638 [72] (the Equifax vulnerability described in §2) and CVE-2017-9791 [73]. In both cases, PHOSPHOR detected the attack string entering the sensitive sink (OGNL in this case).

Proposed Work. *2.1.1: Finding sources and sinks.* In order to perform taint tracking, we need to start out with a pre-defined list of untrusted sources and sensitive sinks. AMPLIFY will be generic, applicable to systems beyond simply HTTP servers, and hence, will need to have a comprehensive and generic mechanism for tracking user-controlled inputs through a variety of applications. A sound but imprecise definition for taint sources will consider all external inputs received by the system under test as untrusted sources. I will refine this list of sources to include only those inputs that flow into the application over the network. Also, by default, data read from files will be considered as potentially user-controlled sources. I will build on my prior work (Pebbles, OSDI 2014), which maintained fine-grained taint tags for files [92], in order to track (non-user controlled) data that is written by the application to files and later read back.

For some applications, purely generic sinks may be valid (e.g. JVM-provided methods that directly execute commands in a shell), though applications that make use of different kinds of parsers might require specialized sinks. Thankfully, while there are a very large number of Java-based applications, there are a relatively small number of (shared) components used by these applications that might compile and execute code (e.g. OGNL, in the case of Struts). I will perform a literature search of known code injection vulnerabilities in Java applications in order to detect common sinks. I will also conduct an extensive evaluation to detect previously unknown sinks in applications by systematically generating attack probing inputs (2.1.4) and detecting when these attacks appear successful. If the attack appeared successful but didn't flow into a previously-known sink, then we have discovered a new sink (or sinks). This process is expected to be more time consuming than the normal operation of AMPLIFY (since it requires testing all possible sources, rather than only sources that flow into known sinks), but will only be performed irregularly.

2.1.2: Prioritizing likely vulnerabilities. With a list of sources and sinks, AMPLIFY will monitor dataflow while each test runs. After tests run, AMPLIFY will likely have a large list of observed (potentially vulnerable) data flows. Although each violation represents a real data flow from an untrusted source I to a dangerous sink O some of these violations may be benign. AMPLIFY will prune this list to contain only true vulnerabilities by systematically attempting to generate exploits (see 2.1.4). This last step will be key to ensure that AMPLIFY only reports real vulnerabilities. A violation might be benign because: (1) The source I is in fact trusted, (2) The sink O is not dangerous, (3) There is a sanitizer that ensures that the data flowing into the sink O is not dangerous. AMPLIFY will consider a variety of heuristics in order to prioritize its vulnerability search. For instance, some sinks (like an expression parser) might be higher risk than others (like file I/O), suggesting that they should be tested first. Other heuristics might include the number of assignments and transformations that occur between the input arriving in the system and reaching the vulnerable sink. An input that flows (untransformed) into a sink might be riskier than another value which is derived from that input and flows into a sink. As part of this work, I will invent new heuristics and evaluate their relative efficacy in detecting both previously-disclosed and new vulnerabilities (see evaluation plan).

2.1.3: Evaluating taint propagation policies. Key decisions when performing taint tracking are: should taint tags be propagated only through data flow, or through data and also control flow? Should tainting an object transitively taint its fields? How (if at all) should taint tags be propagated through exception control flow? Changing how taint tags are propagated can have a significant impact on the resulting relationships. Traditionally, taint tracking is performed through data flow operations only: tags are not propagated through control flow [85]. This limitation can cause false negatives, where taint tags are (incorrectly) not propagated. However, propagating tags through control flow might lead to a loss of precision. There has been no study or evaluation of the gains in recall or losses in precision when using different taint propagation rules, and it's unclear what approach be best for AMPLIFY. I will test different propagation policies and measure the resulting performance of AMPLIFY, optimizing the propagation policies for code injection detection. I will measure precision and recall both at the level of the taint tracking system (e.g. using a benchmark like DroidBench [11]) and also at the full-system level, measuring the vulnerabilities detected with each configuration. Measuring recall will be tricky due to lack of ground truth: I will use a set of known vulnerabilities, plus all of AMPLIFY's vulnerabilities that I manually confirm as a baseline for recall.

2.1.4: Generating Exploits. The end result of AMPLIFY's run will be a list of true vulnerabilities, each accompanied by a test case (attack) demonstrating it. Once vulnerable information flows are detected and prioritized, AMPLIFY will re-execute the test cases that generated these flows, substituting the user-controlled inputs with possible attacks. Note that again, of course, since AMPLIFY has modified the input to the test, the developer-provided test oracles may not hold, in which case AMPLIFY will disable them, running the test with *only* its dataflow oracles. I will build a database of possible attack strings, mapping them to the sinks that they target, and AMPLIFY will leverage this database to test the system under test's resilience. I will generate this database of attack inputs by surveying previously disclosed attacks, and create new attack strings by hand and perhaps using automated concolic execution (Thrust 2). This methodology will allow AMPLIFY to detect new, previously undisclosed vulnerabilities and is based on the insight that most successful injection attacks share common attack strings. Since applications often have multiple sources that flow to the same vulnerable sink, even if a vulnerability was disclosed and patched, the same attack string might be successful on that same (patched) application when applied to a different input. For instance, returning again to Apache Struts and its OGNL parser, a sample attack string might be "`${40+2}`". If this string flows into the OGNL parser (and is evaluated), the output page will contain the text `42`, while if it is not evaluated, the output page might contain the original input, `${40+2}`. Hence, AMPLIFY will be able to automatically determine if it was successful in an attack or not by checking the output of the system. Moreover, this attack string is likely to work on *any* input that flows into the OGNL parser, and mirrors the manual approach that penetration testers often use to probe for vulnerabilities.

Evaluation Methodology. During its development, I will evaluate AMPLIFY's ability to detect injection vulnerabilities on a synthetic benchmark (e.g. a test suite). I will apply AMPLIFY to previously disclosed injection vulnerabilities in Java applications like Struts, Geronimo, Tomcat and JBoss. I have extensive experience running different testing tools with these projects (my prior work has studied the test suites of nearly 100 different open source projects [17, 19, 20, 60]), and am certain that I will be able to integrate PHOSPHOR and AMPLIFY with them easily. I am quite certain that I will be able to find and reproduce existing injection vulnerabilities — there are a wealth of proof-of-vulnerability exploit scripts available for relevant exploits [46], and as a pilot study, I reproduced 5 injection exploits in Struts. As an ultimate goal, I hope to also detect *new*, previously undisclosed vulnerabilities. I will follow best practices for each new vulnerability found, first disclosing it to the vendor, and not publishing details of the exploit until the vendor can release a patch. I will also measure the cost (in terms of execution time and developer/analyst time) of applying AMPLIFY to each of these projects. I will measure the relative precision and recall of AMPLIFY when using different propagation rules with a hand-labeled set of true vulnerabilities.

2.2 Thrust 2: Exploring More Inputs with Concolic Execution

AMPLIFY will also increase the *breadth* of existing test cases by applying novel fuzzing-based techniques to developer-written tests in order to increase coverage of potentially vulnerable information flows. AMPLIFY will leverage ideas from fuzz testing [30, 31, 34, 38, 53, 67, 77, 78, 87, 93, 105, 105] to extend existing test cases to represent more of the diverse inputs that users could make to a server. For instance, in the case of the Struts/Equifax vulnerability described in §2, AMPLIFY would take an existing test which makes a *valid* request and modify it to have an invalid `Content-Type` header, allowing the vulnerability to be discovered through taint tracking.

Concolic execution is a form of fuzzing that concretely executes the system under test, collects path constraints, and then generates new inputs that explore new paths (effectively using symbolic execution). Due to the technical complexities of collecting path constraints and re-executing software with different inputs, existing concolic execution tools such as KLEE, DART, CUTE and JPF-Symbc do not truly concretely execute the system under test, but rather run it in a specialized interpreter that simulates the CPU/language runtime, allowing for easy tracking of constraints and comparison of program states [34, 52, 65, 86, 87]. The speed of collecting constraints and executing new inputs is often cited as the limiting factor in concolic execution, as opposed to the speed of constraint solvers, as reported by the state-of-the-art SAGE project at Microsoft [32]. In contrast, AMPLIFY's concolic execution will allow applications to execute at near-native speeds while tracking constraints, using off-the-shelf JVMs. Re-executing applications and comparing program states with the different inputs that SMT solvers generate can also be a massive slowdown compared to native execution (particularly for systems like DART [52], Klee [34] and JPF-Symbc [65]). Instead, AMPLIFY will leverage my recent advances in lightweight checkpoint and rollback in order to efficiently explore the different program states under different inputs *without* a (slow) specialized interpreter or needing to repeatedly execute the application from the beginning for each input [24]. Although I will implement AMPLIFY for the JVM, the approach that I will take is sufficiently general that it will be possible to apply it to other popular managed languages such as JavaScript, Python, Ruby.

Preliminary Results. Typical fuzzers assume that each input tested executes independently from all that

precede it. If this assumption does not hold, then the fuzzer may discover some input that crashes the system on a fuzzing run but not in isolation; thus limiting the usefulness of such techniques because the crash can not be easily reproduced. The alternative is to restart the system under test between each input tested, however this approach can be very slow. The same isolation problem plagues developer-written tests as well, and I am very familiar with building systems to detect [19, 48] and isolate [17] them. For automated fuzzing, my solution is CROCHET, which will allow AMPLIFY to start the system under test only once, and then *checkpoint* its state, feed it some inputs, and then *rollback* the entire state to its starting state [24]. Like PHOSPHOR, CROCHET functions entirely through bytecode rewriting, utilizing a novel heap traversal algorithm that can efficiently and lazily checkpoint and rollback an entire heap with very low overhead (often twice as fast as the state-of-the-art CRIU [41] system). As a small pilot study, I integrated CROCHET with a simple blackbox fuzzer and measured the improvement of using CROCHET to isolate different fuzzing runs compared to restarting the system under test between each fuzzing test. The entire fuzzing run takes 42 seconds to execute when running all commands on the same server (no restart), 78 when restarting it between commands (an increase of 89%), and just 44 seconds when isolating each input using CROCHET. I expect that this approach will significantly increase the rate at which AMPLIFY can check new inputs. AMPLIFY's path constraint collection will be based on my PHOSPHOR system (described in §2.1).
Proposed Work. *2.2.1: Collecting constraints.* AMPLIFY will use PHOSPHOR to "symbolicate" concrete variables, associating (initially) unbounded symbolic expressions with these concrete variables. AMPLIFY will use PHOSPHOR to record the execution constraints that it witnesses applied to each input of interest. PHOSPHOR already rewrites all bytecode operations to propagate taint tags: AMPLIFY will customize this behavior so that the symbolic expression is refined through execution. PHOSPHOR can propagate any arbitrary metadata as this taint tag, and hence, AMPLIFY will use PHOSPHOR to associate a symbolic expression with each variable, and then override each operation that could combine two of these expressions (e.g. integer operations such as addition) to track the abstract operations that occur on each expression. At branches, AMPLIFY will examine the taint tag being branched on (representing the symbolic branch condition) and record the corresponding constraint. With each expression, AMPLIFY will also record the exact point in program execution that the expression originated, allowing this value to be substituted for a different (generated) concrete value in later exploration. AMPLIFY will support constraints on integer, real, and string variables as well as arrays and objects; it will store these constraints internally using the "Green" symbolic expression package [103]. Once collected, AMPLIFY will continue to use the same Green API to simplify, canonize, negate and solve these constraints using the Z3 constraint solver [43].
2.2.2: Search heuristics and detecting important branches. With constraints collected, AMPLIFY will be ready to begin generating new inputs to direct program execution down new paths. As in traditional concolic execution, by negating individual constraints and attempting to then solve the resulting formula, AMPLIFY will obtain new inputs. However, since I am considering non-trivial programs, I fully expect to be unable to explore the *entire* input space, however my goal is to *test* for vulnerabilities (as opposed to prove the absence of vulnerabilities). Thus, I will use heuristics to guide AMPLIFY's exploration towards branches that may be likely to help expose code injection vulnerabilities. For instance, a static taint tracking approach [11, 101] might reveal that only a fraction of all possible paths allow data to flow from one of the sources to an interesting sink. In this case, AMPLIFY could prioritize its execution to these paths. AMPLIFY might also prioritize sinks based on a risk factor: a sink that directly executes code may be more immediately dangerous than one which saves data to a file (although that file might later be executed). It will be important to tune these heuristics using known vulnerabilities.

Once a new input is available, AMPLIFY will re-execute the original test that the new input was generated from, substituting the generated input for the original. For instance, if the input being perturbed is a byte array — a network request entering a server — AMPLIFY will replace the actual network request with the generated request. I will build on CROCHET, allowing AMPLIFY to efficiently explore many inputs *without* fully re-executing each test, perhaps avoid a relatively slow restart of the application.

CROCHET enables very lightweight incremental checkpoint and rollback within a running JVM: an application like AMPLIFY can request a checkpoint at any time, and then later request a rollback, which will revert all state within the JVM to the previous checkpoint. I will use CROCHET to generate a lightweight checkpoint at each point that AMPLIFY might want to return to (in order to try a different input). CROCHET does not yet support nested checkpoints: repeated calls to checkpoint overwrite the prior saved data. To best support AMPLIFY (which might need to have multiple checkpoints stored at any given time), I will extend CROCHET to allow for these multiple-checkpoint semantics. CROCHET's checkpoint algorithm is thread-safe, and mostly non-blocking: it will be challenging to invent a new algorithm for lazy checkpointing that will be as performant for multi-threaded code (lock free), but also retain multiple checkpoints.

2.2.3: Handling external APIs. AMPLIFY's whitebox approach requires only access to bytecode (not sourcecode), and hence, is applicable out-of-the-box to third party libraries. However, AMPLIFY will not be able to trace dataflow or constraints in non-Java code, which might be *native* code (binary code linked directly into the JVM), or more broadly, external APIs (including web services). Since AMPLIFY is based entirely on concrete execution, it should always be able (in the worst case) to simply repeatedly execute the external API or native routine with different testing inputs in order to observe the output. However, if this external code is *stateful*, then the execution of one test input might impact AMPLIFY's execution of a subsequent input. This same problem impacts both regular (developer-written) tests and automated tests.

Such tests are often said to be *flaky* [20, 68], because when their order is changed, tests might change their outcome due to the invalid state of the environment [17, 19, 108]. Developers commonly resolve these issues by writing pre-test and post-test methods that ensure that the environment is valid before and after a test run. Based on my significant experiences in handling and resolving these test-order dependencies in developer-written tests [17, 19, 21], I will automatically leverage these existing setup and teardown methods that commonly exist in tests in order to clear this external state. Hence, when repeatedly re-executing some narrow part of a test (through checkpoint and rollback), AMPLIFY may need to also execute (at least part of) these setup and teardown methods. AMPLIFY will identify if the code being executed involves external resources (e.g. if it makes calls outside of the JVM), and if so, will perform slicing [58, 104] on the setup and teardown methods to find a slice of code that accesses (and hence may reset) those external resources.

2.2.4: Integration with random fuzzing. While AMPLIFY is primarily designed to perform whitebox fuzzing (i.e. based on collected path constraints), some applications may call for less-directed (blackbox) fuzzing. One typical limitation of whitebox fuzzers is that they may get stuck with constraints that are non-trivial to solve, and hence require spending large amounts of time in the constraint solver. Whitebox fuzzers can also get stuck generating new inputs that do not reveal new program behavior, although they do explore new paths. For instance, for-loops in particular can create *path explosion*, where there are a huge number of paths through that code segment (depending on the number of times the loop is executed), although each path might be nearly equivalent. In contrast, mutation-based fuzzers are "black box," using no knowledge about how that input flows through the program in order to detect vulnerabilities. For instance, AFL mutates each input in isolation through uses bit-flips at random offsets, addition and subtraction of small integers, and insertion of interesting integers (*e.g.,* 0, 1, -1, `INT_MAX`, etc) [105].

AMPLIFY will leverage mutation-based fuzzing in two ways, depending on the relative speed of constraint solving to program execution, which will vary from system-to-system, and likely from input-to-input. If AMPLIFY detects that far more time is being spent solving constraints than testing new inputs, then AMPLIFY will resort to (fast) mutation-based fuzzing while waiting for a response from the constraint solver. If instead, constraint solving takes *less* time than testing those inputs, then AMPLIFY will use a fitness function inspired by mutation-based fuzzing in order to prioritize which of those inputs to test first, for instance, prioritizing those that might be more likely to reach dangerous sinks.

2.2.5: Distributed constraint collection. For simplicity of discussion, I have assumed that AMPLIFY was applied only to applications that execute in a single process on a single computer. However, distributed systems are increasingly common, where an application is split up into multiple components that are deployed as different processes communicating over a network. In this configuration, AMPLIFY will run in a peer-to-peer mode (using Zookeeper for coordination [9]), aggregating the data flow and path data from each node in the system. AMPLIFY will intercept all network messages being sent and received by the system under test: if it detects that two components of the system are communicating with each other, it will embed and extract dataflow information. In addition to propagating metadata between processes, AMPLIFY will also need to coordinate checkpoint and rollback between multiple processes. Maintaining synchronization between all of these components will be the main challenge of this task: different components might interact using different protocols, and it is possible that not all components will be observable by AMPLIFY. I will investigate different consistency and communication models, evaluating their tradeoffs in performance and correctness for AMPLIFY. To simplify implementation, I will assume that all participating nodes run in a JVM (rather than in different platforms).

Evaluation Methodology. I will evaluate AMPLIFY's concolic execution component at both the component and system level, measuring the speed of vulnerability detection, speed of input generation, cost of computation, and developer effort. At the component level, I will compare AMPLIFY's performance in generating new test inputs to state-of-the-art JVM-based test input generation tools, including JPF-Symbc [65], jCUTE [86] and EvoSuite [47], using similar workloads that those tools have been evaluated on. At the system level, I will evaluate how rapidly AMPLIFY detects code injection vulnerabilities using different heuristics for its concolic exploration. I will measure the cost of running AMPLIFY's concolic execution engine in terms of CPU-time, again comparing it with prior work. Finally, I will evaluate AMPLIFY's applicability to distributed systems by testing it on both distributed and monolithic applications.

2.3 Thrust 3: Detecting Vulnerabilities in Continuous Integration and Regression Testing

Developing software is a continuous process, with developers constantly adding new features and fixing bugs. Rather than be designed to run in a "one off" model, I plan for AMPLIFY to be tightly integrate into a modern continuous-integration (CI) based development environment, where developers change code and correspondingly run regression test suites to detect newly-introduced faults. CI services automatically build and test code, often for each commit that a developer pushes to their version control repository. Hence, there is a wealth of additional data that can be made available to tools (e.g. the results of these tools on prior executions of the project). At the same time, CI services are generally somewhat resource-constrained: developers want results of a CI run quickly [61], and hence, it's often not possible to directly integrate heavyweight static and dynamic analysis tools with CI executions. I will investigate new approaches to apply tools like AMPLIFY in CI, leveraging historical execution data to find more vulnerabilities faster. I will develop integrations for AMPLIFY with popular build systems (such as Maven [8] and Gradle [1]), as well as with popular CI systems like TravisCI [2].

Preliminary Results. I have significant experience building new approaches and tools for regression testing, in particular, in a CI environment. For example, my recent DEFLAKER work considered the automated detection of flaky tests (tests that can nondeterministically pass or fail even for the same version of the same code) in regression testing environments [20]. Unlike prior approaches for detecting flaky tests that relied purely on information about the "current" version of code being tested, DEFLAKER also uses historical test results from CI. As part of my evaluation, I integrated DEFLAKER directly with the CI pipeline for 96 open source projects that used the freely available service *TravisCI* [2]. DEFLAKER found 87 previously unknown flaky tests, and developers generally found my reports to be useful. My prior work has also examined how statement coverage changes over time in projects that use CI [60].

Proposed Work. *2.3.1: Incremental test amplification.* One commonly used approach for accelerating regression testing is *regression test selection* (RTS), where for each revision of the system being tested, only tests that might be impacted by recent changes are executed [51, 59, 79, 80, 83]. This reduces turnaround time as developers wait for results. If a traditional RTS approach determines that a given test should be executed (e.g. that its outcome might have changed), AMPLIFY will perform its own analysis to determine if it should try to amplify that test (generating new inputs), or if this amplification is unlikely to detect new vulnerabilities. When AMPLIFY is used in its incremental mode, it will consider (1) the prior paths that were explored in each test and the results of that exploration and (2) the changes to the system under test since the last execution. Specifically, AMPLIFY will combine its information about which statements were covered by each test (and each amplified test) with the set of which statements were changed since its last execution. If a path does not cover changed statements, then its outcome could not have changed. In my prior work, DEFLAKER [20], I introduced the notion of *hybrid differential coverage*, a technique that tracks which (recently changed) statements are executed by each test. DEFLAKER's differential coverage is *hybrid* in that it automatically changes granularity (from statement to classfile-level tracking) depending on the kinds of changes that were made to each file: some changes impact execution beyond a single statement (e.g. changing the super type of a class can change dynamic method dispatch).

2.3.2: Collaborative amplification. I will also design AMPLIFY to optimize repeated executions of the same tests on the same revision of code. In particular, in scenarios where developers run tests locally on their own machines, and then *also* run them on shared build servers (e.g. in a CI infrastructure), AMPLIFY will centrally cache its results, reducing repeated effort and parallelizing its load where possible. A key challenge here will be ensuring that path constraints are sufficiently normalized such that they can be easily transplanted between multiple executions. For instance, if AMPLIFY is run with a limited time budget on a developer machine and then later runs (on that same exact version of the system under test) on a build server, AMPLIFY would recognize exactly what was checked in the first run in order to avoid redundancy. In its normal operational model, AMPLIFY doesn't need to correlate constraints observed on inputs from multiple executions of the same test/system under test. However, in this case, it will need to have a stable mapping from symbolic variables across executions (enabling it to perform this matching). I will investigate different techniques for assigning stable names to each symbolic variable, perhaps based on stack traces and invocation counts or the shape of the constraints. If necessary I will also investigate compression techniques (based on the assumption that many constraints will overlap between different paths).

2.3.3: Differential constraint analysis. When a test fails, it is generally indicative of (1) a fault that the developer recently introduced, or (2) flakiness in the test, which might cause it to pass or fail non-deterministically. Broadly, fault localization techniques generate a list of likely portions of program code that are the cause of that test failure. I propose to leverage AMPLIFY's path constraint data to create a new form of fault localization, which will be useful for debugging the vulnerabilities that AMPLIFY detects, as well as general faults. Specifically, given the path constraints from a passing execution of a test and the path constraints from a failing execution of the same test, AMPLIFY will determine the key branching condition(s) that caused the failing test execution to diverge, and eventually to fail. My intuition is that failing test assertions are likely failing due to the system under test's processing of some input, which then is propagated through control and data flow to an assertion. Prior related techniques for fault localization have relied on backwards program slicing to create a slice of statements that are required to reach the test failure. Instead, by analyzing the path constraints applied to each input, AMPLIFY will detect exactly where in the system that input was processed differently (between the passing and failing test). Combining this information with traditional backwards slicing can further demonstrate the differences between the two executions [104].

Evaluation Methodology. I will evaluate the improvements in vulnerability detection rates using AMPLIFY's incremental and shared CI configuration, measuring the reduction in analysis time compared to running AMPLIFY "from scratch" on each revision of projects. I will evaluate the applicability of AMPLIFY at the system-level in CI environments by deploying it in the CI runs of hundreds of existing open source projects, a methodology that I pioneered with DEFLAKER [20]. For each CI build of each target application, I will run my own version of that build in the same CI environment, collecting statistics on the relative slowdown imposed by AMPLIFY (compared to the normal build). I will also perform large-scale controlled validation of AMPLIFY's scalability, measuring how useful its incremental analysis mode and constraint cache are in reducing execution time, again reusing methodology from DEFLAKER. For DE-

FLAKER's evaluation, I used cloud resources to quickly evaluate it on some 6,000 versions of 26 projects at a very modest cost of only several hundred dollars. I will evaluate AMPLIFY's fault localization on prior vulnerabilities, comparing its reported localization to the true location (identified by the location of the developer fix). I will evaluate it on traditional faults using the popular Defects4J benchmark, which contains hand-labeled faults and developer-written test cases that expose them [63].

2.4 Thrust 4: Production Monitoring and Defense

Thrusts 2 and 3 will result in a vast database of the observed constraints that are applied to different inputs as they flow through the target application. AMPLIFY will take advantage of its accumulated database of path constraints in order to deploy extremely lightweight telemetry that will detect when an application receives an input matching different constraints from what was tested, and hence, might represent a new attack. Alternatively, if AMPLIFY detects that an input will expose behavior that *has* been seen before, then it may be able to apply optimizations to the application based on those prior executions which might help offset any performance slowdowns that AMPLIFY otherwise imposes. The core idea is that many different inputs likely expose some common core of program behavior. For instance, there are inputs that lead to exceptions, and others that do not. If many inputs all satisfy a core set of constraints, then AMPLIFY can easily detect abnormal (untested) inputs in the field — those that do not meet these constraints.

Preliminary Results. This thrust will also build extensively on PHOSPHOR (as described in §2.1) and CROCHET (as described in §2.2). I expect that my experiences building tools to help developers diagnose field failures (Chronicler [26] and Parikshan [10]) will help inspire my approach to gathering telemetry from production applications. Both of these systems are *record and replay* based, which is to say, they record non-deterministic events when an application runs, allowing that non-determinism to be automatically replayed while running the same code in the lab, hence, reproducing the recorded execution. Record and replay-based systems are useful for reproducing bugs that occur in production systems; my goal here is different but related: to identify how two different inputs might result in similar program behavior. In this context, the proposed work is also inspired by my prior work in detecting "code relatives" — two syntactically dissimilar code regions that result in similar behavior (regardless of similarity of inputs) [96].

Proposed Work. *2.4.1: Determining the important constraints.* Rather than force programs to execute inputs that result in *different* constraints (and hence, different program behavior), here, I will consider how to identify the *common* constraints that are shared by most inputs. I expect that many inputs will result in entirely unique path constraints, and hence, will not match previously observed paths. However, I also hypothesize that in many cases, a large class of inputs might share some large set of common constraints. These common constraints represent shared program behavior: even though the path that each input executes might be different, some prefix of that path might be common. For instance, a constraint prefix might indicate that an input is in error, causing an exception, but the complete path constraints of that input might not match all inputs that would result in the same exception. If this is the case, then AMPLIFY will *still* be able to draw upon knowledge from past executions, keeping in mind that these results only apply to that prefix. There may be other shared constraints that are not shared prefixes, but rather represent shared features of these inputs, such as passing some input validation. I will mine shared path constraints from the constraint database, and utilize this data in runtime monitoring and optimization.

2.4.2: Constraint checking in the field. In the testing environment, AMPLIFY tracks the various constraints that are applied to application inputs. In deployed software, AMPLIFY will check each input to see if it matches constraints previously observed. If AMPLIFY detects that an input is likely to cause a failure or attack, it could warn the user immediately, immediately throw an exception, add additional tracing/logging code for debugging, or perhaps even try to recover gracefully [84, 89]. Alternatively, if AMPLIFY detects that an input doesn't match prior constraints, it might be configured to perform dynamic taint tracking on that input in order to ensure that it doesn't result in a code injection attack. This database of previously-seen constraints may be updated over time based on observations obtained from developers' testing, and also from other executions in the field (described further in §2.4.4). Along with the set of constraints witnessed, AMPLIFY will record high-level fault data (e.g. whether or not the input triggered a failure). Checking that an input satisfies a set of constraints is a relatively fast operation, and I hypothesize that in most cases, inputs will match constraints witnessed by other executions. If an input matches a previously seen execution, then AMPLIFY will consider leveraging that information to perform program optimizations that may help mask any potential overhead that it otherwise might introduce.

2.4.3: Optimization. AMPLIFY will use its profiling data to optimize the program's execution to reduce observed overhead from its monitoring. While the JVM performs aggressive optimizations, there are limits to the level of optimization that the JVM will make on a method. For instance, by default, the JVM will never attempt to inline a method if that method is larger than 35 bytes [76]. Inlining is a key optimization that allows far more aggressive optimization to take place, but the downside to inlining a large method is that it can cause code bloat, and if the target method to be inlined requires dynamic dispatch, then the JVM must insert a guard in the optimized code to ensure that the caller type matches what was inlined. Both of these problems can be solved based on path profiling, allowing the JVM to do its job better. In the event that an input matches previously seen constraints, AMPLIFY will consider using the historical data to optimize the system's handling of that input. One example optimization might be to specialize methods, removing (potentially) large portions of methods that AMPLIFY knows apriori will not be executed (because it knows the path that the system will take to process the input), making the method smaller and more amenable to inlining. Further, AMPLIFY might be able to replace (expensive) dynamically dispatched method calls with (cheap) statically dispatched method calls, since again, AMPLIFY knows from the path profile data the correct receiver type for each method call. Each of these optimizations will require a fast analysis in order to determine if applying the optimization is worthwhile: there will be a fixed cost to applying the optimization (and deoptimizing it if necessary). In remaining with the spirit of the rest of my approaches, I will implement these optimizations *without* modifying the JVM. I expect to implement these optimizations using a novel variant of my heap traversal and code-patching algorithm that I first prototyped for CROCHET [24].

2.4.4: Constraint recording on/off. If an input *doesn't* match previously seen constraints, then AMPLIFY might record the path constraints on that input for future analysis. Of course, this will be dependent on the end-user's privacy requirements, since even path constraints can contain sensitive information [35, 40]. Prior approaches that recorded path constraints in field executions relied on automatically *replaying* the execution (in a separate process than the user is interacting with) in order to collect the constraints due to the immense overhead of collecting them [35, 40]. These prior approaches were limited by the way that they collected constraints: using heavy-weight symbolic execution engines like KLEE or JPF. Since AMPLIFY's constraint recording is based on concrete executions, I am convinced that in the hopefully infrequent case that an input doesn't match previously seen constraints, it will be able to capture them online. The core challenge here will be devising a technique to dynamically turn constraint recording on and off, so that there will be no overhead when an input matches the previously-seen constraints, and then minimal overhead when doing the recording. I will invent code transformation techniques so that analyses like AMPLIFY's constraint recording (and others, more generally) can easily be flipped on-and-off, drawing on my extensive experiences building runtime systems for the JVM [16, 17, 19, 24].

Evaluation Methodology. I will evaluate AMPLIFY's detection of vulnerabilities "just in time" by comparing the constraints collected from vulnerabilities (as explored in thrusts 1 and 2) with constraints collected from benign executions. I will evaluate AMPLIFY's ability to detect important constraints in both attack and benign inputs by measuring the proportion of different inputs marked as similar, comparing its results to other execution similarity approaches, such as code relatives [96], using again data collected from running AMPLIFY on real vulnerabilities. I will evaluate the performance of AMPLIFY's field-based constraint checking, optimization and constraint collection by using the DaCapo macro benchmark suite [29], building a database of constraints from the test suite of each of the apps included in the benchmark, and then evaluating AMPLIFY's performance on the benchmark's workload.

Thrust	Y1	Y2	Y3	Y4	Y5
1. Depth: Testing for Code Injection Attacks with Dynamic Taint Tracking	●	●	○	○	○
2. Breadth: Exploring More Inputs with Concolic Execution	○	●	●	○	○
3. Crosscutting: Detecting Vulnerabilities in Continuous Integration and Regression Testing	○	○	●	●	○
4. Crosscutting: Production Monitoring and Defense	○	○	○	●	●

Table 1: Tasks to be addressed in each of the five years

3 Research Timeline

Table 1 shows a high level overview of the five-year research plan. This project will involve one funded PhD GRA; I plan for a second PhD GRA to participate in this work as well, who will be financially supported by the GMU CS department (see chair letter), I also expect to involve several unfunded "project students" (who will work on the project in exchange for course credit). I will begin with both students working on the taint tracking aspects. In the second year, one student will continue to focus on taint tracking aspects, while the other will begin work on concolic execution. In year three, one student will continue the concolic execution work while the other integrates the components with the continuous integration model. The entire team will work together on the crosscutting aspects in years four and five. This schedule recognizes dependencies between tasks, in particular the construction of the taint tracking and concolic execution components.

4 Broader Impacts

Societal and Technical: Software greatly affects society as it is increasingly relied upon in healthcare, finance, national infrastructure and defense. The line between high-assurance and general-purpose software is increasingly blurred, as nowadays nearly *any* insecure software can have severe economic consequences. This project will develop, validate and disseminate better tools that any engineer can use to detect code injection vulnerabilities in their applications during testing. Moreover, this project will target vulnerabilities in software running in the JVM, a platform which has increasingly been a target of attack, in including the high-profile Equifax attack. The resulting methods and tools developed in this project can be used as a platform for future research on other kinds of vulnerabilities in JVM-based applications.

Dissemination: I will disseminate results in the form of research papers, educational material, and open-source software. I will have a particular emphasis on releasing open-source software, building on my strong record [27] with released tools used in academia and even in industry, *e.g.,* VMVM [17] was integrated into a product line at ElectricCloud [44] which led to follow-up work [19,21,22]; Phosphor [14,16,18] has been adopted and extended by a variety of other researchers [4,5,42,90,100]. I expect that each core component of AMPLIFY will be a useful tool for enabling future research in complimentary directions. For instance, the taint-tracking components could be used to assess other features of test cases (such as brittleness [62]) and the concolic execution component could be used for other input generation tasks.

Educational: I will continue to engage high school, undergraduate, and graduate students in research: since beginning at GMU, I have supervised two high school, five BS, one MS degree and one PhD student in research projects. My curriculum design efforts for CS 475 (described below) were recently recognized by the students who unanimously rated me 5/5 for the evaluation question "My overall rating of the teaching."

Undergraduate Curriculum Development: I volunteered to take ownership of GMU's undergraduate concurrent and distributed systems class, CS 475 [13], which I first offered in Spring 2018. All CS undergraduates at GMU are required to pick one of three electives, including 475, and hence, 475 is taken by many students (the class is full every semester). I completely overhauled the syllabus, bringing it up to speed with state-of-the-art research concepts. I brought several research topics into the classroom through lectures: I used several academic papers (including GFS [50]) as case-studies, where the students used their foundational knowledge of distributed systems to see how the various design choices made in these large systems were made. I also encouraged the students to use a research-based Java race detection tool, RVPredict, in their projects [81], which they responded quite positively to. My course materials are all publicly available [13], and I reach out to colleagues who teach similar classes to swap notes and stories. In the future, I will include CROCHET as a case study of advanced concurrency control topics like optimistic locking and compare-and-swap, which are used extensively in CROCHET's thread-safe algorithm. I also plan to include at least one entire lecture on code injection attacks, which are very relevant to distributed systems.

Graduate Curriculum Development: I will translate the software engineering and systems building lessons from building AMPLIFY into modules and assignments for my seminar, CS/SWE 724: Program Analysis for Software Testing [15]. First offered in Fall 2017 as a "special topics" seminar, it is now an official course, regularly offered and listed in the university's course catalog. In the beginning of the semester, I have several assignments that get the students' feet wet in program analysis tools while also helping them to scope their term projects. In Fall 2017, these assignments involved code coverage and dynamic taint tracking and led to several successful student-driven projects related to fault localization, flaky test detection and taint tracking. One (otherwise terminal-degree) MS student decided to join me as a PhD student to further pursue the topic. In future offerings, include at least one assignment focused on constraint tracking and solving and will encourage students to pick projects in vulnerability detection.

Broadening Participation in Computing. I strive to create a welcoming, inclusive environment that broadens opportunities for all students, including underrepresented groups. I have successfully mentored underrepresented students at all levels and will continue engaging undergraduate, full-time MS, and workforce MS, as well as PhD students in research. My first two PhD students are both from underrepresented groups, and I intend for both to work on the projects described in this proposal. I serve as advisor to GMU's Student-Run Computing and Technology (SRCT) group, an undergraduate organization that supports students interested in computing [69]. SRCT has a very diverse membership, and is committed to including all interested students regardless of gender, race or sexual orientation. I have and will continue to co-organize the NSF-supported undergraduate Programming Languages Mentoring Workshop at SPLASH, an event that focuses on broadening the participation of underrepresented groups in PL and SE research [25,82]. In 2017, PLMW attendees were evenly split by gender (24 male, 24 female); 10 were Hispanic, African American, Native Hawaiian or Pacific Islander. I am considering organizing a local version of this workshop.

Evaluation of BPC activities. While BPC is a long-term effort, with major results (*e.g.,* overall impact of my efforts on the field) not becoming clear for some time, I will continuously evaluate and reflect on my BPC activities. I will measure success through a variety of personal and professional development factors, including those recommended to me by my university's Office of Students as Scholars [49]. For instance, I will consider undergraduate mentorships successful if the mentees become more interested in pursuing a graduate education in computing, and if they feel more comfortable and confident in their pursuit of computing. For more structured activities (e.g. the PLMW workshop series), I will continue to work with CRA's Center for Evaluating the Research Pipeline (CERP) to track outcomes both in the short term (just after the workshop) and in the long term (years after the workshop). In these contexts, I've been looking at metrics such as: an increased understanding of the field, an increased understanding of academia, an increased desire to apply to/remain in graduate school/faculty positions (as appropriate).

5 Results from Prior NSF Support

PI Bell has one NSF grant recently awarded but not yet started, CCF-1763822 "SHF: Medium: Collaborative Research: Enhancing Continuous Integration Testing for the Open-Source Ecosystem" (total: $399,591, October 1, 2018 – September 30, 2022). **Intellectual merit:** This project addresses the problem of regression testing in the new setting of CI. Traditional solutions for the problem do not provide optimal tradeoffs in modern software. This soon-to-start project will develop practical techniques and tools for enabling faster and more reliable CI testing. **Broader impact:** More effective CI testing will provide faster and better feedback for developers, resulting in better software delivered to customers faster. This project will result in publicly available tools that can benefit everyday software developers. This CAREER proposal will amplify existing developer-written tests by adding new oracles and new inputs, which is complementary to my ongoing efforts to increase the reliability and speed of existing tests in the SHF-medium grant.

References

[1] Gradle. http://gradle.org/.

[2] Travis CI. https://travis-ci.org/.

[3] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[4] S. Amir-Mohammadian. *A Formal Approach to Combining Prospective and Retrospective Security*. PhD thesis, University of Vermont: Computer Science, 2017.

[5] S. Amir-Mohammadian and C. Skalka. In-depth enforcement of dynamic integrity taint analysis. In *ACM Programming Languages and Security Workshop (PLAS)*, 2016.

[6] Apache Foundation. Apache struts code coverage. https://coveralls.io/github/apache/struts.

[7] Apache Foundation. Apache struts release history. https://struts.apache.org/releases.html.

[8] Apache Software Foundation. The apache maven project. http://maven.apache.org/.

[9] Apache Software Foundation. Apache zookeeper. http://zookeeper.apache.org.

[10] N. Arora, J. Bell, F. Ivancic, G. Kaiser, and B. Ray. Replay without recording of production bugs for service oriented applications. In *33rd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2018, 2018.

[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[12] J. Bell. CROCHET. https://github.com/gmu-swe/crochet.

[13] J. Bell. Cs 475 spring 2018 - concurrent & distributed systems. http://www.jonbell.net/gmu-cs-475-spring-2018//.

[14] J. Bell. Phosphor. https://github.com/Programming-Systems-Lab/phosphor.

[15] J. Bell. Program analysis for software testing, fall 2017. http://www.jonbell.net/swe-795-fall-17-program-analysis-for-software-testing/.

[16] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, October 2014. ACM.

[17] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. In *36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, New York, NY, USA, June 2014. ACM. ACM SIGSOFT Distinguished Paper Award.

[18] J. Bell and G. Kaiser. Dynamic Taint Tracking for Java with Phosphor (Demo). In *International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 409–413, New York, NY, USA, July 2015. ACM.

[19] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient Dependency Detection for Safe Java Test Acceleration. In *10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 770–781, New York, NY, USA, September 2015. ACM.

[20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: Automatically detecting flaky tests. In *Proceedings of the 2018 International Conference on Software Engineering*, ICSE 2018, 2018.

[21] J. Bell, E. Melski, M. Dattatreya, and G. Kaiser. Vroom: Faster Build Processes for Java. In *IEEE Software Special Issue: Release Engineering*. IEEE Computer Society, March/April 2015.

[22] J. Bell, E. Melski, G. Kaiser, and M. Dattatreya. Accelerating Maven by Delaying Test Dependencies. In *3rd International Workshop on Release Engineering*, RELENG '15, page 28, Piscataway, NJ, USA, May 2015. IEEE Press.

[23] J. Bell, C. Murphy, and G. Kaiser. Metamorphic Runtime Checking of Applications Without Test Oracles. *Crosstalk the Journal of Defense Software Engineering*, 28(2):9–13, Mar/Apr 2015.

[24] J. Bell and L. Pina. Crochet: Checkpoint and rollback via lightweight heap traversal on stock jvms. In *Proceedings of the 2018 European Conference on Object-Oriented Programming*, ECOOP 2018, 2018.

[25] J. Bell, B. Ryder, J. Vitek, and S. Nadi. Splash 2018 pl/se mentoring workshop (plmw). `https://2018.splashcon.org/track/splash-2018-PLMW`.

[26] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight Recording to Reproduce Field Failures. In *International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, May 2013. IEEE Press.

[27] Software and Datasets from Jonathan Bell's Group. `http://www.jonbell.net/software`.

[28] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[29] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmark suite. `http://www.dacapobench.org/benchmarks.html`.

[30] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. ACM.

[31] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[32] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[33] S. Bousquet. Criminal charges filed in hacking of florida elections websites. `http://www.miamiherald.com/news/politics-government/article75670177.html`.

[34] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[35] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 319–328. ACM, 2008.

[36] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications*, ISCC '06, Washington, DC, USA, 2006. IEEE.

[37] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *2009 ACM Workshop on Secure Web Services*, SWS '09. ACM, 2009.

[38] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[39] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07*. ACM, 2007.

[40] J. Clause and A. Orso. Camouflage: Automated Anonymization of Field Data. In *33rd International Conference on Software Engineering*, ICSE '11, pages 21–30, New York, NY, USA, 2011. ACM.

[41] J. Corbet. Checkpoint/restart (mostly) in user space. *LWN.Net*, 2011.

[42] J. Cox. *Improved Partial Instrumentation for Dynamic Taint Analysis in the JVM*. PhD thesis, UCLA: Computer Science, May 2016.

[43] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] ElectricCloud. DevOps Automation and Continuous Delivery solutions to accelerate application build, test and release. `http://electric-cloud.com/`.

[45] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, Berkeley, CA, USA, 2010. USENIX Association.

[46] Exploit Database. Offensive Security's Exploit Database Archive. `https://www.exploit-db.com`.

[47] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[48] A. Gambi, J. Bell, and A. Zeller. Practical test dependency detection. In *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, ICST 2018, 2018.

[49] George Mason University. Office of students as scholars. `https://oscar.gmu.edu/faculty-staff/assessment/`.

[50] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[51] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[52] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[53] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[54] J. Goldman. Researchers find russian hacker selling access to u.s. election assistance commission. `https://www.esecurityplanet.com/hackers/researchers-find-russian-hacker-selling-access-to-u.s.-election-assistance-commission.html`.

[55] Google. Error-Prone: Catch Common Java Mistakes as Compile-Time Errors, howpublished=`https://github.com/google/error-prone`.

[56] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference*, ACSAC '05, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[57] W. G. J. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *SIGSOFT '06/FSE-14*, pages 175–185, New York, NY, USA, 2006. ACM.

[58] C. Hammacher. JavaSlicer. `https://github.com/hammacher/javaslicer`.

[59] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, pages 312–326, 2001.

[60] M. Hilton, J. Bell, and D. Marinov. A large-scale, longitudinal study of test coverage evolution. In *33rd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2018, 2018.

[61] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *31st IEEE/ACM Conference on Automated Software Engineering (ASE 2016)*, pages 426–437, Singapore, Singapore, Sept. 2016.

[62] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, 2014.

[63] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[64] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[65] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[66] T. Kitten. Card fraud scheme: The breached victims. `http://www.bankinfosecurity.com/card-fraud-scheme-breached-victims-a-5941`.

[67] LLVM Project. libfuzzer - a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[68] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, pages 643–653, Hong Kong, China, Nov. 2014.

[69] Mason SRCT. Student-Run Computing + Technology. `https://srct.gmu.edu`.

[70] W. Masri and A. Podgurski. Using dynamic information flow analysis to detect attacks against applications. In *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems&Mdash;Building Trustworthy Applications*, SESS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[71] R. Miller. "foreign" hack attack on state voter registration site. `http://capitolfax.com/2016/07/21/foreign-hack-attack-on-state-voter-registration-site/`.

[72] National Vulnerability Database. Cve-2017-5638 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-5638`.

[73] National Vulnerability Database. Cve-2017-9791 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-9791`.

[74] National Vulnerability Database. National vulnerability database search for "execute arbitrary commands". `https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&query=execute+arbitrary+commands&search_type=all`.

[75] Open Web Application Security Project. `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf`.

[76] Oracle. Jvm options. `http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html`, 2013.

[77] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY, USA, 2017. ACM.

[78] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, Feb. 2017.

[79] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *ICSM*, pages 358–367, 1993.

[80] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[81] Runtime Verification, Inc. Runtime verification predict. `https://runtimeverification.com/predict`.

[82] B. Ryder, L. Pollock, and J. Bell. Splash 2017 pl/se mentoring workshop (plmw). `https://2017.splashcon.org/track/splash-2017-PLMW`.

[83] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE*, pages 46–53, 2001.

[84] T. Saoji, T. H. Austin, and C. Flanagan. Using precise taint tracking for auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 15–24, New York, NY, USA, 2017. ACM.

[85] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[86] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *CAV*, pages 419–423, 2006.

[87] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[88] A. Seshagiri. How hackers made $1 million by stealing one news release. `https://www.nytimes.com/2015/08/12/business/dealbook/how-hackers-made-1-million-by-stealing-one-news-release.html?_r=0`.

[89] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 37–48, 2009.

[90] J. Singleton. *Advancing Practical Specification Techniques for Modern Software Systems*. PhD thesis, University of Central Florida: Computer Science, 2018.

[91] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *CCS '13*, New York, NY, USA, 2013. ACM.

[92] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained Data Management Abstractions for Modern Operating Systems. In *11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 113–129, Berkeley, CA, USA, 2014. USENIX Association.

[93] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Computer Security Applications Conference*, pages 477–486, 01 2008.

[94] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA '11*. ACM, 2011.

[95] D. Staahl. Hack that targeted arizona voter database was easy to prevent, expert says. `http://www.azfamily.com/story/32945105/hack-that-targeted-arizona-voter-database-was-easy-to-prevent-expert-says`.

[96] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara. Code Relatives: Detecting Similarly Behaving Software. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, November 2016. Artifact accepted as platinum.

[97] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan. Identifying functionally similar code in complex codebases. In *Proceedings of the 24th IEEE International Conference on Program Comprehension*, ICPC 2016, 2016.

[98] F.-H. Su, J. Bell, C. Murphy, and G. Kaiser. Dynamic inference of likely metamorphic properties to support differential testing. In *Proceedings of the 10th International Workshop on Automation of Software Test*, AST 2015, 2015.

[99] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, pages 85–96, New York, NY, USA, 2004. ACM.

[100] J. Toman and D. Grossman. Staccato: A Bug Finder for Dynamic Configuration Updates. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[101] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.

[102] University of Maryland. FindBugs - Find Bugs in Java Programs. `http://findbugs.sourceforge.net`.

[103] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[104] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[105] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[106] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 595–605, Piscataway, NJ, USA, 2012. IEEE Press.

[107] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, 23(4):32:1–32:28, Sept. 2014.

[108] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA*, pages 385–396, 2014.