Security Vulnerability Verification through Contract-Based Assertion Monitoring at Runtime

by

Alexander M. Hoole
B.Sc., University of Victoria, 2003
M.A.Sc., University of Victoria, 2006

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Security Vulnerability Verification through Contract-Based Assertion Monitoring at Runtime

by

Alexander M. Hoole
B.Sc., University of Victoria, 2003
M.A.Sc., University of Victoria, 2006

Supervisory Committee

Dr. I. Traore, Supervisor
(Department of Electrical and Computer Engineering)

Dr. T.A. Gulliver, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. K.F. Li, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. J. Weber, Outside Member
(Department of Computer Science)

Supervisory Committee

Dr. I. Traore, Supervisor
(Department of Electrical and Computer Engineering)

Dr. T.A. Gulliver, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. K.F. Li, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. J. Weber, Outside Member
(Department of Computer Science)

ABSTRACT

In this dissertation we seek to identify ways in which the systems development life cycle (SDLC) can be augmented with improved software engineering practices to measurably address security concerns that have arisen relating to security vulnerability defects in software. By proposing a general model for identifying potential vulnerabilities (weaknesses) and using runtime monitoring for verifying their reachability and exploitability during development and testing reduces security risk in delivered products.

We propose a form of contract for our monitoring framework that is used to specify the environmental and system security conditions necessary for the generation of probes that monitor security assertions during runtime to verify suspected vulnerabilities. Our assertion-based security monitoring framework, based on contracts and probes, known as the Contract-Based Security Assertion Monitoring Framework (CB_SAMF) can be employed for verifying and reacting to suspected vulnerabilities in the application and kernel layers of the Linux operating system. Our methodology for

integrating CB_SAMF into SDLC during development and testing to verify suspected vulnerabilities reduces the human effort by allowing developers to focus on fixing verified vulnerabilities. Metrics intended for the weighting, prioritizing, establishing confidence, and detectability of potential vulnerability categories are also introduced. These metrics and weighting approaches identify deficiencies in security assurance programs/products and also help focus resources towards a class of suspected vulnerabilities, or a detection method, which may presently be outside of the requirements and priorities of the system.

Our empirical evaluation demonstrates the effectiveness of using contracts to verify exploitability of suspected vulnerabilities across five input validation related vulnerability types, combining our contracts with existing static analysis detection mechanisms, and measurably improving security assurance processes/products used in an enhanced SDLC. As a result of this evaluation we introduced two new security assurance test suites, through collaborations with the National Institute of Standards and Technology (NIST), replacing existing test suites. The new and revised test cases provide numerous improvements to consistency, accuracy, and preciseness along with enhanced test case metadata to aid researchers using the Software Assurance Reference Dataset (SARD).

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My wife Lindsey, son Liam, daughter Abby, and our extended family and friends, for their support, encouragement, and love throughout this journey.

My supervisor, Dr. Issa Troare, along with my supervisory committee and other professors, colleagues, fellow graduate students, and co-authors, for their mentoring, support, encouragement, collaboration, and patience.

Natural Sciences and Engineering Research Council of Canada (NSERC), Mathematics of Information Technology and Complex Systems (MITACS), and the University of Victoria for scholarship funding.

Microdev Engineering and HPE Security Fortify for the valuable work and research experience as well as the academic license provided by Fortify, through Brian Chess, starting in 2007.

Aurelien Delaitre, Charles de Oliveira, and Frederick Boland from National Institute of Standards and Technology (NIST) for their collaboration resulting in replacement tests suites 100 and 101 which began in May of 2014 and completed April of 2015. Specifically, Aurelien's contributions to Appendix C and his updates to the numerous test cases, as well as Charles' contributions to help enable automation noted in the appendix under Section B.4 and updates to the metadata on the SARD site which add value to the security assurance community. It was truly a pleasure working with both researchers.

Do the right thing. It will gratify some people and astonish the rest.
Mark Twain

DEDICATION

To Lindsey, Liam, and Abigail.

Chapter 1

Introduction

New vulnerabilities are continually uncovered, and systems are configured or used in ways that make them open to attack.
-Dorothy Denning

Vulnerability, as defined by Oxford Dictionaries Online[1], indicates that an entity is *susceptible to physical or emotional attack or harm*. Weakness is defined as *a quality or feature regarded as a disadvantage or fault*. Finally, a monitor is defined as *an instrument or device used for observing, checking, or keeping a continuous record of a process or quantity*.

In the domain of application security, weaknesses are the bugs in software implementation, or code, that could lead to a system being vulnerable to attack if not addressed (a *potential vulnerability*). Vulnerabilities are those weaknesses (or flaws) related to architecture, design, implementation and configuration that can be directly exploited during execution to gain access to a system. The existence of an exploitable weakness, once discovered, improves the likelihood that a malicious hacker can launch a successful offense. The identification, verification, and removal of weaknesses during the system development life cycle is part of having a strong defense. While many different approaches to detecting software weaknesses and vulnerabilities exist, preemptively identifying, verifying and correlating vulnerabilities with their underlying weaknesses is still a challenge. Monitoring during runtime is one approach that can assist in the verification, and application of potential preemptive actions, of

[1] https://en.oxforddictionaries.com

potential vulnerabilities for identified weaknesses.

1.1 Context

Security has always been a hybrid combination of art and science as throughout history humans have attempted to protect valuable assets. Our modern information driven society has placed an increased value on data as well as the transfer and storage of information. In the last decade, industry and academia have pushed for more secure solutions for information technology assets and facilities due to a rise in malicious hacking and security threats. During the same time, systems have been moving away from being based solely on proprietary technologies to include implementations based on common and open computing techniques and standards. As a result, the risk exposure of these systems to attacks and piracy has increased considerably.

Many different approaches have been presented recently toward solving the problem of weak security through preventative and reactive measures; however, we obviously have not yet found the solution since security related attacks continue to persist.

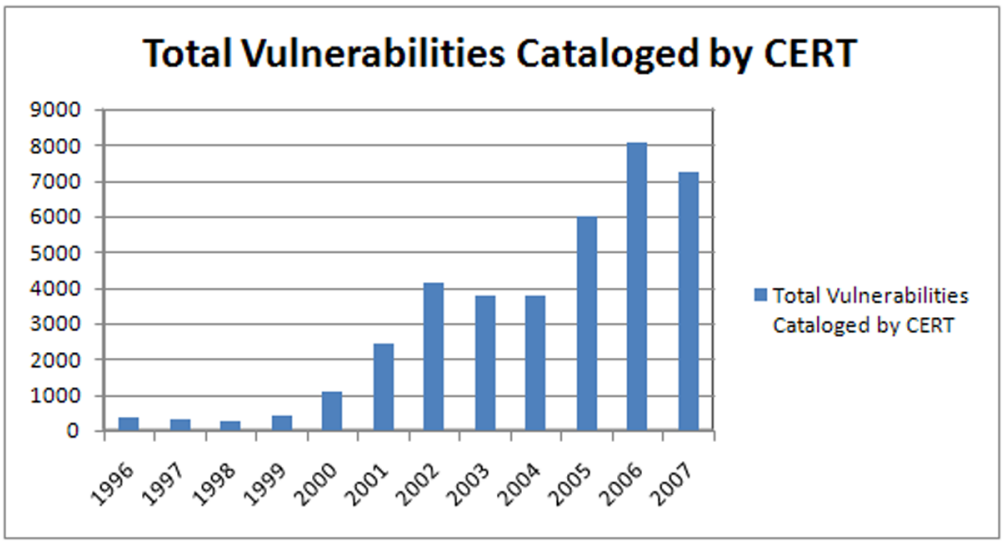

Figure 1.1: CERT vulnerability statistics.

From 1995 till 2008 the Computer Emergency Response Team (CERT) tracked the number of security vulnerabilities reported and cataloged through their coordination center. In their effort to reduce risk in existing systems and the number of new vulnerabilities, CERT performs vulnerability remediation. Reported vulnerabilities,

depicted in Figure 1.1, have continued to increase.[2] The trend has continued since 2008, even though CERT no longer provides the above statistics, we can follow certain trends via the National Vulnerability Database (NVD) where vulnerabilities have been tracked since 1988.[3] Figure 1.2 depicts current statistics up until July 2017. With 7,049 vulnerabilities recorded in the first half of 2017, the current year already represents the second highest in recorded history. This increase in reported vulnerabilities could be the result of a more security conscious software community, increased volume of created software, bug bounty programs, or any number of other reasons. Regardless of the underlying reasons, the fact remains that more secure software systems need to be created with fewer exploitable vulnerabilities. If secure systems were the norm we would not be observing this alarming trend.

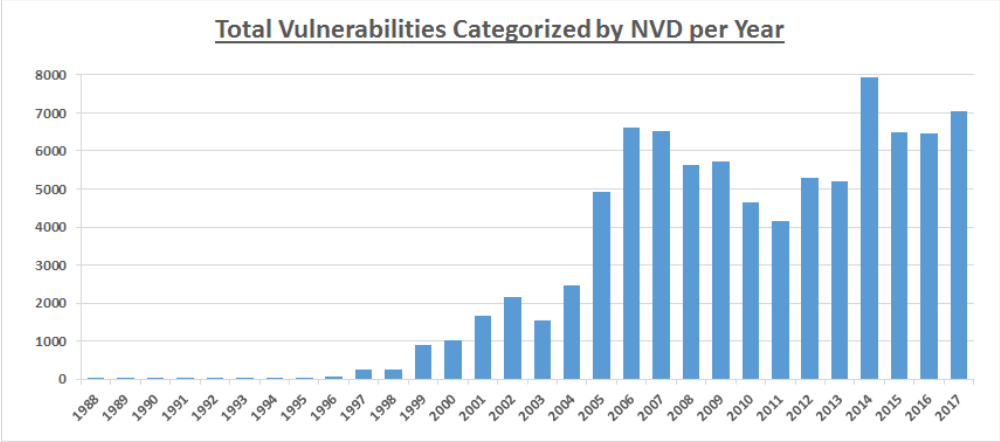

Figure 1.2: National Vulnerability Database statistics (as of July 3rd, 2017).

Gary McGraw identifies three trends influencing the growth and evolution of the software security problem [1]. First, *connectivity* to the Internet has increased the number of attack vectors and the ease of which an attack can be made. Second, *extensibility* of software is allowing systems to grow in an incremental fashion which potentially adds new security vulnerabilities to existing systems. Lastly, the extensive increase of software *complexity* in modern information systems leads us to a greater number of vulnerabilities. These three trends will continue and lead us to one, hopefully obvious, conclusion. Security vulnerabilities must be preemptively identified and resolved during design and testing before being released to the general public.

In the last decade, we have observed a promising shift in industry and academia

[2]Complete statistics were available until 2008 at *http://www.cert.org/stats/fullstats.html*.
[3]Statistics available at the following URL: *https://web.nvd.nist.gov/view/vuln/statistics*.

to reduce security vulnerabilities during the software development life cycle (SDLC), rather than attempt to patch the problem after software is shipped [1, 2, 3, 4, 5]. The report of a single vulnerability has a multiplicative effect since every system that includes the affected version(s), of the vulnerable software component for a particular vulnerability, is effected and each consumer of that system is further affected. If we can reduce *security defects* during the SDLC we reduce not only the number of vulnerabilities but also the risk of attack.[4]

Dan Geer wrote a column discussing different approaches to measuring latent zero-day vulnerabilities using metrics borrowed from biology [6]. Geer reminds us of common approaches such as *capture-recapture* and *removal-capture* using the example of "How many frogs are in the pond?" as a specialization of the general question of the form "How many X are there in Y?". This question can also take the form of "How many exploitable weaknesses are there in this piece of software?". In order to answer such a question, we would need to have detection strategies for identifying and verifying weaknesses and a way to measure the effectiveness of such an approach. Geer also refers to a question posed by Bruce Schneier: "Are vulnerabilities in software dense or sparse?" The answer to this question helps frame the importance of the work of finding weaknesses. If weaknesses in software artifacts are extremely common, or dense, then each identified and fixed has minimal impact. If weaknesses are less common, then their removal has a strong impact. The question of vulnerability density in software is a little more complex than the above question indicates. Just as there are many different types of animals living in a rain forest, there are many different types of weaknesses that can exist in a software artifact. In the same analogy, not all animals are considered to be as dangerous as others in a particular environment. For example, in the context of predators, an eagle is considered less dangerous than a shark when the prey is 15 meters underwater. As such, perhaps the question needs to be more context specific: "Are the vulnerabilities that we need to focus on for our particular piece of software, in its given execution environment, dense or sparse?" Ultimately, Schneier states that "[w]e also need more research in automatically finding and fixing vulnerabilities, and in building secure and resilient software in the first place" [7].

The remainder of this chapter outlines the security activities still lacking in most

[4]Security defects can be divided into two categories [1]. First, *security bugs* are an implementation-level software problem, such as a buffer overflow. Second, *security flaws* are not only visible as an implementation problem, they are also visible (or not) at the design level, such as an improperly implemented overridden function.

SDLCs, the research problem, our expected contributions to the field of application security and software engineering, followed by an outline of this dissertation.

1.2 SDLC and Security

Security policy documents are often used by organizations to specify the laws, rules, practices, and principles that govern how to manage, protect, and transfer sensitive information. These policy documents represent a cornerstone from which software requirements can be built. Requirements in turn drive most modern software/system development life cycles. During the SDLC there are many opportunities to mitigate security vulnerabilities.

An SDLC is in many cases an iterative and recursive process that clearly identifies the stages that should lead a successful software project through its entire development life cycle.[5] We are interested in integrating security into every phase of the SDLC. In fact, several tools and methodologies have already begun to integrate themselves accordingly.[6] For example, the Building Security In Maturity Model (BSIMM)[7], Program Review for Information Security Management Assistance (PRISMA)[8], and Software Assurance Maturity Model (SAMM)[9] are all examples of approaches to improve processes around security during system development life cycles. These approaches include, among other things, facets of security policies, measurement, education, procedures, requirements, architecture, implementation, review, test, and integration. We believe, however, that there is a great deal of work remaining in this area.

The SDLC is still lacking sufficient models, methods, and tools that assist in creating more secure and reliable software products. The intended audience for this work includes individuals and teams fulfilling the following roles during a SDLC: analyst, architect, developer, tester, maintainer, user, and support. Essentially, all of the development-related stakeholders in the SDLC.

Serpanos and Henkel asserted that a unified approach to dependability and security assessment will let architects and designers deal with issues in embedded computing

[5]The individual phases of the SDLC should be known to most readers; however, should further explanation be required it is thoroughly documented in other literature.

[6]NIST on integration into the SDLC http://csrc.nist.gov/groups/SMA/sdlc/index.html.

[7]https://www.bsimm.com

[8]http://csrc.nist.gov/groups/SMA/prisma/index.html

[9]http://www.opensamm.org/

platforms [8]. The observation that security and dependability are interrelated is an important one. Serpanos and Henkel differentiate the two as security flaws being problems that are exploited on purpose, while flaws that are exploited by accident would be qualified as dependability problems. It would be interesting to have a framework that can be used for both dependability and security. Thus, we have kept dependability in mind while designing our framework; however, we have maintained our focus on security vulnerability monitoring since it is our primary target.

The goal of our research is to create new methods, models, and tools that integrate with existing phases of an SDLC to create more secure software. We cannot depend on the consumer to have sufficient protection mechanisms in place on his/her systems.[10] We need to have a better defensive strategy and take a more active role during development to ensure software has fewer security vulnerabilities from the start.

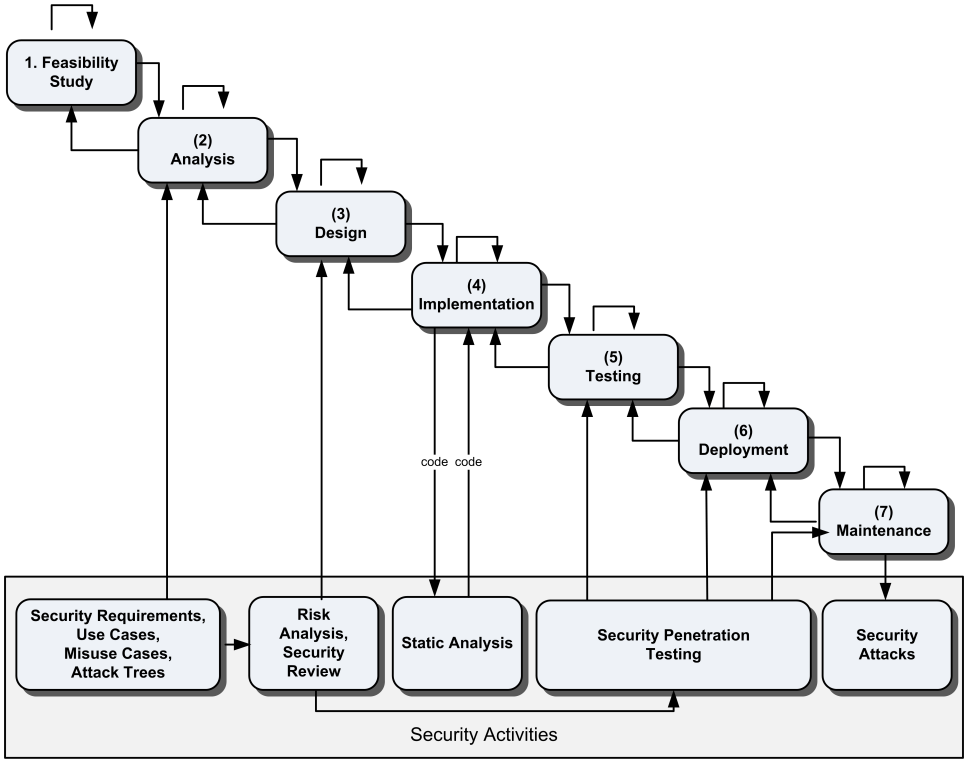

Figure 1.3: Security activities integrated into the typical waterfall SDLC. Regular SDLC steps are numbered and linked diagonally to security activities displayed horizontally.

A modified form of the SDLC is depicted in Figure 1.3 showing how various security

[10]Consumers deploy Intrusion Detection Systems (IDS), firewalls, and other products to help reduce security risks.

activities can be integrated into the iterative and recursive SDLC. Existing SDLC hybrids integrate some of the steps identified in Figure 1.3 such as those put forward by CERT, Microsoft's Michael Howard and Steve Lipner [2], and others. Nothing has been identified to date that guarantees security in software systems; however, our aim is to help reduce risk associated with the presence of security vulnerabilities.

1.3 Research Problem

Software systems, containing security vulnerabilities, continue to be created and released to consumers. We need to adopt improved software engineering practices to reduce the security vulnerabilities in modern systems. These practices should begin with stated security policies and end with systems that are quantitatively, not just qualitatively, more secure.

There are at least three vectors at play in trying to optimize security vulnerability mitigation. Each of these vectors provides opportunities for continuous improvement.

1. ***Identification*** of potential vulnerabilities using approaches such as code review, static analysis, dynamic analysis, and penetration testing. Striving for high true-positive, low false-positive, low false-negative, and high true-negative identification of vulnerabilities.
2. ***Verification*** of suspected vulnerabilities to verify reachability/exploitability. Identifying a subset of weaknesses that have been confirmed as exploitable vulnerabilities can help prioritize remediation activities.
3. ***Remediation*** of vulnerabilities by fixing coding/design errors, configuring environment/context, or deploying countermeasures.

Existing approaches for identifying, monitoring, and verifying security vulnerabilities are still insufficient. In 1998, Voas *et al.* identified the verification of exploitability of identified vulnerabilities through the program's standard input by an attacker as an important remaining research problem [9]. In recent years, we have begun to see industrial strength utilities to assist in the identification and removal of security vulnerabilities [5]. Static analysis tools provide a promising way to identify many known vulnerabilities by their underlying weaknesses; however, they also show the need for ways to test if a potential vulnerability is actually exploitable. Our CB_SAMF

could provide a means to validate identified vulnerabilities and thus contribute further to the set of growing SDLC tools.

While static approaches such as taint analysis can identify potential vulnerabilities, verification can only occur during or after execution. Black *et al.* recently published a report to the White House Office of Science and Technology Policy indicating that present approaches, while having made great progress, are still insufficient [10]. A methodology and tool set for improving security during development and testing that can span the multiple layers of software in modern systems is still needed. Any new approach should also provide a means of measuring the improvement in security.

We have observed widespread application of perimeter defenses, such as firewalls and IDSs, intended to stem the damage caused to consumer systems. This approach of deploying reactive measures will never completely address security issues because it fails to remove the underlying problem of vulnerable software containing defects. IDSs have been implemented as monitoring frameworks; however, IDSs do not fix software vulnerabilities, they only track and potentially prevent them from being exploited [11, 12, 3, 13, 14, 15, 16, 17, 18]. Specifically, we still require better tools and methodologies to identify, reduce, and remove security defects in software [10].

Software monitors (or *monitors* for short) have been used for real-time systems to ensure proper behavior [19, 20, 21]; however, most approaches do not allow for the addition of relevant fields needed by the monitoring framework in order to identify security vulnerabilities and react to them [11, 12, 22, 23, 24, 25]. Other approaches to vulnerability detection in software, such as static analysis methods, are able to identify many known vulnerabilities via their underlying weaknesses; however, they tend to suffer from a high rate of false-positives.

The challenge in this work is to create artifacts that provide a process and model for identifying potential weaknesses, and set of tools that will assist developers produce more secure and reliable products, by verifying suspected vulnerabilities in a measurable way. In particular, we desire to remove security vulnerabilities during development/implementation and testing rather than depend on reactive operational tools such as firewalls and IDSs. Our aim is to integrate the new artifacts into the SDLC during the development/implementation through testing phases, although, it is also possible to use a hybrid of this approach to monitor applications during operation in a production environment (deployment and maintenance).

1.4 Proposed Approach

Mechanisms for detecting security weaknesses in code suffer from false positives and lack the ability to verify exploitability. We propose a model for identifying and verifying weaknesses combined with a contract-based security assertion monitoring framework (CB_SAMF) for measurably reducing the number of security vulnerabilities that are exploitable, across multiple software layers. We also introduce measurements and metrics to track improvements to security assurance when the model and framework is applied to an enhanced SDLC. We show in this work how CB_SAMF can be integrated into a development life-cycle to validate suspected vulnerabilities in the application layer, the Linux kernel, and related device drivers.

While the notion of contracts is not a new idea in software engineering, when we began our work, contracts had not been applied as a means for ensuring security properties through runtime monitoring [26, 27, 28, 29, 30]. Recently, however, several approaches have begun to attempt to address security vulnerabilities using runtime monitoring [31, 24]. Contracts can provide a useful mechanism when applied toward identification and tracking of vulnerabilities. We propose a form of binding contract that binds two or more parties to perform, or not perform, a set of actions. Using such a contract we will be able to bind caller(s) and callee(s) to deal with issues involving timing, property values, and other events.

We chose the notion of contracts for an assertion framework so that we can state precise properties about a system without having to modify the code directly using a separation of concerns approach. In order to precisely state properties about a system, we must be able to express specific predicates in the context of the code and its runtime environment. For the purpose of rigor, it is desirable to have a formal specification of the desired properties referenced in the predicates of the contract, since these properties translate to executable code that must execute within the context of the running process when the relevant probe is inserted dynamically into the system (without recompiling the code). An example set of common security problems found in systems, and targeted by our framework, includes the following groupings:

- Exploitable Logic Error
- Inadequate Concurrency Control
- Weak Dependencies/Altered Files
- Inadequate Parameter Validation (Incomplete/Inconsistent)
- Inadequate Authentication/Authorization/Identification
- Implicit Sharing of Data and Data Leakage

Exploitable logic errors are difficult to track down; however, if we can identify environmental, historical, or timing information related to the expected behaviour,

contracts can be written to detect misuse. Parameter validation issues can be handled by pre and post conditions. Concurrency, accountability, and protocol issues can be tracked through the use of historical, environmental, pre and post conditions. Finally, the addition of historical and environmental assertions should allow us to track vulnerabilities related to weak dependencies and data leakage.

Our monitoring framework, through the use of kernel-based probes in Linux, allows us to inject the probe and monitoring logic directly into the context of the running application rather than require a separate process. This is akin to injecting a forced moral consciousness into the application which requires the process to be honest about its behavior. As a live, or online system, our contract-based assertion monitoring framework immediately reports violations when they occur and permits reactive measures. This contrasts with a large number of monitoring systems that do offline processing of event logs. Contract-based approaches which are offline, such as the MONPOLY Usage-Control Policy monitoring tool presented by Basin *et al.*, that specify linear temporal logic policy statements (similar to contracts) are not able to perform reactive measures. MONPOLY also does not appear to allow for the definition of probes to extract the necessary event data for verification of security vulnerabilities (i.e. it assumes the event mechanism and associated logs are provided) [32]. Furthermore, offline contract-based models should evaluate performance metrics on the union of augmented runtime performance of the instrumented application and the offline processing of event data in order to understand impact. Finally, these offline monitoring systems are designed for forensics investigation and passive compliance verification rather than online verification or prevention of vulnerabilities. To be capable of verifying security vulnerabilities during testing requires an inline or online system that is capable of accessing, manipulating, and storing metadata for the properties relevant to the vulnerability category. As such, we do not attempt to implement an offline runtime verification system [32].

The content of this dissertation combines and expands both the notion of contracts in our framework and the integration of our framework into a modified SDLC. We must deal with complex architectures where support must be provided at the hardware, operating system, device driver, and application layers. In particular, the ability to monitor layers beneath the application layer and our focus on removing vulnerabilities before products are delivered to clients provide added benefits.

Our proposed monitoring framework can be integrated early during SDLC. A security policy document is often used as part of the processes identifying the secu-

rity requirements. Security requirements are then used during the identification of misuse cases (along with normal use cases) that are intended to identify potential vulnerabilities. Once prioritized, these misuse cases can then drive the creation of attack trees which further identify intrusion scenarios. The intrusion scenarios can then be used during design and testing to create sequence diagrams and associated test cases. Finally, during implementation, sequence diagrams and static analysis reports can be generated to identify *potential security vulnerabilities* (for example, system/function calls that have known vulnerabilities). Once a potential vulnerability has been identified, a "contract" can be created using assertions and additional rules to guard against, or verify, a given vulnerability. Potential vulnerabilities, such as those identified through static analysis and code review, should be verified before resources are consumed to remove them. These contracts can then in turn be used to generate security probes that are used during execution to track forensic data in our monitoring framework (CB_SAMF) to verify the suspected vulnerabilities.

Consideration is also given as to whether output formats from existing weakness identification tools, such as static analysis tools, may be translated into a format that may be used by the assertion monitoring framework. Ultimately though, the focus of this work is on how to identify suspected vulnerabilities, verify they are exploitable/reachable via the creation and consuming of contracts, generation of assertion probes, monitoring assertions, and reacting appropriately using the monitoring framework. Finally, focus is also given on how to measurably improve security assurance processes/products used during an improved SDLC by performing the above steps over time.

1.5 Research Contributions

The focus of this work was to create a model for identifying potential weaknesses, monitoring suspected vulnerabilities in applications, for newly discovered potential security weaknesses, during the implementation through testing phases. The following key contributions are made in this dissertation:

1. Methodology, and SDLC integration strategy, for identifying potential vulnerabilities. This contribution has been published in one conference [29] and submitted to a journal that is currently under revision [30].

2. Contract-based model for runtime security assertion monitoring that verifies vulnerability reachability and exploitability. This contribution has been published in two conference workshops [26, 27] and one journal [28].

3. New security metrics and measures intended for the weighting, prioritizing, determination of confidence, and detectability of potential vulnerability categories. This contribution has been submitted and currently under revision in one journal [30].

4. New security assurance test suites[11], developed in collaboration with NIST. Test cases provide improvements to consistency, accuracy, and preciseness along with enhanced test case metadata to aid researchers using the Software Assurance Reference Dataset (SARD). This contribution has been published in one conference [29].

1.6 Dissertation Organization

The rest of the thesis is structured as follows:

Chapter 2 presents related work in the areas of monitoring and intrusion detection, contracts, static and dynamic analysis, and concludes with a review of the context of our proposed framework.

Chapter 3 provides a background on weaknesses and vulnerabilities in software systems and introduces our general approach to weakness identification and verification.

Chapter 4 introduces our Contract-Based Security Assertion Monitoring Framework (CB_SAMF) along with its syntax, semantics, and an exploratory case study.

Chapter 5 reviews existing metrics for assessing a security assurance product before introducing new measures and metrics for customizing assessments to stakeholder requirements and for improving security assurance programs over time.

Chapter 6 discusses various software security taxonomies and test suites before presenting our empirical evaluation results for verifying exploitable vulnerabilities when

[11] Test Suite 100 (https://samate.nist.gov/SARD/view.php?tsID=100) and Test Suite 101 (https://samate.nist.gov/SARD/view.php?tsID=101).

provided weakness assertions. Experiments also demonstrate how security metrics, defined in Chapter 5, when applied to inaccurate test suites lead to flawed evaluations and how new security metrics can lead to improved security assurance practices.

Chapter 7 Summarizes the contributions of this work, provides concluding remarks, and discusses possible avenues for future work.

Chapter 2

Related Work

> *Security is always going to be a cat and mouse game because there'll be people out there that are hunting for the zero day award, you have people that don't have configuration management, don't have vulnerability management, don't have patch management.*
> -Kevin Mitnick

Contracts and monitors have been applied to different software quality aspects and concerns. Contracts allow us to describe and enforce specific rules while monitoring allows us to track the state of a system. In this chapter we review related research for these two concepts and present interesting findings in the reviewed literature relating to monitors and intrusion detection, contracts, and related analysis tools. Furthermore, the third section reviews related research on measures and metrics related to software security assurance. The final section of this chapter will provide an overview of the context of our work.

2.1 Monitors and Intrusion Detection

Monitoring is the practice of observing something with the option of maintaining a recorded history of the said object. Furthermore, monitoring systems have been used for a wide variety of purposes to observe and analyze the behavior of a second system.[1]

[1]Monitoring frameworks are not unique to software systems as they are frequently used in other engineering disciplines to monitor behavior.

Monitors have been approached by Peters and Parnas relating to physical real-time systems [19]. Peters and Parnas assert that requirements-based monitors, derived from the specification of the system, can be used to ensure that real-time systems behave correctly. Their requirements-based approach focuses on environmental, monitored, and controlled state functions to specify the monitor for a system.[2] The distinction between environmental and system state functions for monitoring is an important one in modern system design and implementation, since architectures tend to span multiple software layers. The trend towards more complex systems increases the demand for a monitoring framework which can span all of the software layers rather than a system that targets only a singular component. The approach by Peters and Parnas is prone to both false-positive and false-negative because of device limitations. Modern day software, real-time or not, also needs to be evaluated for correct behavior. We can also apply monitoring frameworks to the area of correct software security requirements through the creation of security contracts.

The approach followed by Pohlack *et al.* is also centered around real-time system monitoring and debugging practices [33]. Their monitoring framework, Ferret, is based on-top of a para-virtualized version of Linux. With Ferret they are able to insert monitoring sensors using a sensor directory, registered monitors, and registered clients resulting in monitoring of a real-time system. Pohlack *et al.* do not focus on security applications for their work; however, they do make use of the Kprobes feature of the Linux kernel to implement several of their features. In comparison to the work of Pohlack, we recommend considering the SystemTap framework to implement monitoring probes rather than direct usage of Kprobes, target security vulnerabilities, and provide reactive rather than only passive measures. In addition, we recommend pro-actively monitoring the system during development rather than on production systems, with the stated goal of releasing more secure systems.

For security systems, several monitoring approaches have been presented that are based on policy driven models such as those by Schneider [34], Chari and Cheng [35], and Zimmerman *et al.* [36]. An alternative approach has been introduced by Ko, Ruschitzka, and Levitt using a specification based intrusion detection system (IDS) approach [11]. Payer introduces a host-based, per-process, anomaly-based IDS technique called HexPADS for detecting attacks by maintaining statistics on existing

[2]Environmental quantities are those that can be observed externally from the system, monitored quantities are those that should affect the behavior of the system, and controlled quantities are those that the system may be required to change the value. A quantity can be environmental and monitored, or controlled and monitored or not monitored at all.

low-overhead, hardware-based, performance counters [16].

Historically, intrusion detection can be divided into two separate categories called *host-based* and *network-based* intrusion detection. Recent advances, however, add the classifications of *knowledge-based* and *hybrid* IDSs [18]. Furthermore, modern day intrusion detection systems are typically classified into the following two separate approaches: *anomaly-based*, and *policy-based* detection [11, 3, 18].

Anomaly-based detection is built on the premise that if a program behaves in a manner other than what we believe to be its normal behavior, then this new behavior is most likely an intrusion [13]. Anomaly-based detection needs to be taught the normal behavior before it can detect an intrusion. Disadvantages of this approach include: the need for training for a baseline, false negatives, and a potentially high rate of false-positives. The main advantage of this approach is that it can detect previously unidentified intrusions.

Monitoring approaches have been presented based on policy driven models for security systems [37, 38, 35, 11, 34, 39, 24]. Policy-based detection can be broken down further into two sub-types referred to as *signature-based* and *specification-based* detection.

Signature-based detection, also referred to as misuse detection, focuses on the creation of signatures that identify known sequences of instructions that lead to an intrusion. This approach suffers from at least two disadvantages. First, a vulnerability must be identified before it can be caught. Second, it can be difficult to write a signature that can catch all variations of a known attack. In contrast, this approach has the advantage of a lower rate of false-positives.

Specification-based detection is typically defined on a per-application basis rather than system-wide signatures (as is often the case with misuse detection). This method of detection determines whether a sequence of instructions violates the specification for a given program or system.

Early work by Ko, Fink, and Levitt [12], motivated by specification-based intrusion detection, focused more on specifying what a program was allowed to do and then monitored the program to ensure conformance based on audit trails. Their approach for automated detection of vulnerabilities in privileged programs (those which have elevated priority) through execution monitoring provides useful insight to several classes of security vulnerabilities. While the notation used for a program policy is straight-forward and based on predicate logic and regular expressions, this early

approach does not appear to handle temporal events, changes in files[3], nor non-privileged programs. The events for the monitor are generated by the system-call layer in UNIX operating systems and does not have the ability to track other events. Each audit event generated for the monitor consists of the following fields: *uid*, *progid*, *op*, *euid*, *egid*, *path*, *pmode*, *ouid*, *ogid*, *devid*, *fileid*, and *port*. This approach is not able to capture all forms of vulnerabilities such as some DoS attacks, race conditions, and design flaws. Ko *et al.* recognize that audit trails have several limitations and raise the issue of being able to track issues in other layers of a software system other than just system calls. Specifically, they state that finding a way to instrument a program automatically to generate audit trails that provide information needed for monitoring would be useful. We believe that the use of a contract-based approach based on the processor breakpoint mechanism could provide a mechanism for system-wide integration of monitors.

A later approach proposed by Ko, Ruschitzka, and Levitt [11] focuses on utilizing security specifications to describe the intended behavior of programs, then during runtime they produce traces of the monitored programs with the ultimate goal of performing real-time intrusion detection. Specifications for each application come in the form of a 'trace policy' that is specified using a *parallel environment grammar* (PE-grammar). Underlying their approach is a scanning mechanism that detects violations by analyzing audit trails during runtime. Ultimately, it is the parsing of the audit trails that is the key to their implementation when it detects operations being performed by users that contradict the trace policies. Ko *et al.* continue their approach by identifying key attributes in program behavior that are important to security. These include the following: access of system objects, sequencing, synchronization, and race conditions. Access to system objects is an important observation since one can determine security violations when a user attempts access to system objects he/she should not be accessing. Sequencing, approaches the issue of time-ordered events which is also important when trying to observe the exploitation of vulnerabilities during runtime. Synchronization (and the special case of race-conditions) attempt to address security issues arising from poor locking and precedence implementations (such as improper identification of a critical section and the use of mechanisms such as mutexes to protect them). One could argue that these are a subset of the attributes

[3]For example, rather than having a rule that monitors only the path of files used by a program, we should also be able to specify attributes related to the file. Suppose a file has been replaced with a malicious version by a trusted user.

that are widely considered necessary in systems requiring reliability and stability. Furthermore, this set of properties and the use of only system-call events still cannot identify all classifications of attack. Specifically, buffer overflows cannot be identified by the above since a locally allocated memory buffer or array does not fall under any of the above categories. In the case of a buffer overflow it might be argued that the target application that is to be executed can be captured. Denial of service also is not covered by the above arguments. If an application completely ties up a resource causing starvation, which it is otherwise able to access under normal circumstances, this approach does not appear to scale. Finally, the claim that all events in the system can be totally ordered may not hold on symmetric multiprocessing (SMP) systems where actions can be performed in parallel.

Gao *et al.* introduce an interesting model for host-based intrusion detection based on program call graphs that they call execution graphs [14]. Their model focuses on the order in which system calls are executed and detect a potential anomaly when a call outside of the expected execution call graph is detected. As indicated earlier in this proposal, not all attacks will be caused by system call sequences. In their analysis Gao *et al.* classify system call monitoring into the following three categories: black-box method in which call traces are extracted from sample runs of the application, grey-box method which also extracts additional information from the process, and white-box method which builds their model using static analysis of either the source or binary files.[4] There is an underlying assumption in their new (grey-box) technique that when training is being done for a given application it is currently in a pristine state. If a system is trained on a modified program the generated traces are not valid to begin with. This model also does not consider the values of arguments passed to the system calls, nor does it consider the call graph that occurs in the operating system or device drivers. Another assumption made by this approach is that a program being analyzed does not have any existing weaknesses that can be manipulated from outside of the program. We would prefer a system that is available at compile time and have a monitoring system available for all instances of programs that do not have source code available. A white-box method would allow the analysis of source code for a wide range of vulnerabilities.

Voas *et al.* created two tool-sets for identifying security vulnerabilities in software

[4]Static analysis is an evaluation performed against source code based on its form, structure, or content. Dynamic, or runtime, analysis is an evaluation performed against a target during execution (http://www.stsc.hill.af.mil/crosstalk/1994/07/xt94d07l.asp).

[9]. The first tool-set was designed to perform dynamic analysis of software using program inputs, fault injection and assertion monitoring called Fault Injection Security Tool (FIST). The second tool-set targeted static analysis of program properties named VIsualizing STatic Analysis (VISTA). Both of these tool sets operated against the C and C++ languages and were to be used during design, implementation and testing phases of the SDLC. Voas *et al.* also identified a major important research problem that remained after their work was completed. Their research was able to discover security vulnerabilities through a process of fault injection and dynamic monitoring; however, the tools were not able to determine whether an attacker could exploit the vulnerabilities by providing standard input through the program interface.

Bhatkar *et al.* [38] furthered the work on control flow of Gao *et al.* and others by adding the ability to track system call arguments as well as system calls. With their added functionality, in essence the ability to also follow data-flows, they are able to identify additional attack types that are not detectable in earlier attempts. We agree that a framework should support access to system call arguments, and further, should provide access to any functional arguments.

Work by Petroni *et al.* observes that intruders are able to hide their presence from compromised systems despite the abilities of the current generation of integrity monitors. In their paper, Petroni *et al.* introduce an architecture for semantic integrity constraints using a specification language-based approach [37].

Barringer *et al.*, conducted further work on runtime monitoring in which they introduced formalisms for defining expressive specifications with parameters resulting in Quantified Event Automata (QEA) [40]. These parametric properties are used to monitor cases where events carry data values. Previous monitoring approaches that could not represent parameters were insufficient for many security verification problems. Their paper divides specification formalisms based upon levels of expressiveness and usability, and monitoring based upon efficiency. Some approaches focus on efficiency (e.g. JAVAMOP [41] and TRACEMATCHES [42]), while others focus on expressiveness (e.g. EAGLE [43], RULER [44], LOGSCOPE [45], and TRACECONTRACT [46]). Since we desire expressiveness for our form of contracts, we build and extend our contract syntax from the grammar from EAGLE and add the notions of context, history, and response [43]. Furthermore, we desired to create an efficient monitoring implementation that instrumented live/inline running systems, rather than require offline, or recompilation-based inline/online, approaches such as those used in the above methods. Meredith *et al.* classify the running mode of monitors as one of the

four following types: *inline* (woven into the code), *online* (running parallel to the code), *outline* (receiving events from the program remotely), or *offline* (analyzing generated event traces from logs) [41]. Most of the above approaches are for offline monitoring of traces and as such are not considered for comparison. For those cases that are capable of doing online monitoring (e.g. EAGLE and RULER), they are limited to Java and thus application space (i.e. not capable of instrumenting kernel-level logic). Our approach to monitoring is for inline instrumentation of both user-space and kernel-space binary applications where we specifically target C-based Linux binary execution.

Through further analysis of these tools and methodologies it may be possible to generate similar models directly from sequence diagrams (similar to the work proposed by Wang *et al.* [47]) during design phase, rather than through static analysis of source code. If this is possible it could allow for the generation of source code "templates" that have reduced security risks before developers begin their implementation work. The output of static analysis tools can also help drive the creation of the sequence diagrams later in development.

Falcone, Currea, and Jaber implement a runtime verification (RV) and runtime enforcement (RE) framework for Android applications [48]. The approach, similar to others above, is based upon refactoring the code to instrument it with monitoring capabilities at compile time using aspect-oriented programming and third party runtime verification products such as Java-MOP and RuleR. The approach is interesting as they implement their architecture using a cloud, or device embedded, approach to refactoring using a custom AspectJ compiler called *Weave Droid*. It is, however, dependent upon decompiling/recompiling code, only works for Android mobile apps, and is limited to Java. While their benchmarking experiment is based upon code that makes extensive use of data structures, they do detect several security vulnerabilities that were previously identified by William Enck *et al.* [49] using static analysis provided by HP Fortify SCA. There is no technological association between the runtime verification of Falcone and static analysis detection conducted by Enck.

Halfond and Orso use an anomaly-based model to approach the single weakness category of SQL Injections (SQLi) by combining static analysis and runtime monitoring to overcome limitations of using either singular approach by itself [50]. After building a conservative model of the legitimate queries that could be conceived by the application using static analysis, the runtime part of their implementation uses monitoring to ensure that dynamically generated queries are compliant with the static model

(deviations are reported as exploits). Static analysis is used first to identify the "hotspots" in the code where SQL queries are issued to a database and then models are built for each hotspot. The source code is then instrumented with calls to monitors that guard the query at runtime. Halfond and Orso have taken an interesting approach to solve a single vulnerability type, however, it is limited to only SQLi, and requires access to, modification of, and recompilation of the source code. In addition, several noted limitations are also made by the authors (scalability and false negatives).

Leucker and Schallhart provide a brief account of *runtime verification* in which they compare against model checking, theorem proving and testing [20]. They specifically define runtime verification as "the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [20]". As such, Leucker and Schallhart introduce runtime verification as an online, or offline, "lightweight verification technique" that focuses only on the detection/satisfaction of violations of correctness properties which can compliment other techniques such as model checking, theorem proving, and testing. *Model checking*, while an automatic verification approach, requires that a precise model of the system be created first. Additionally, model checking focuses on verifying all executions of the system and infinite traces, while runtime verification is producing a verdict on a given trace/execution. *Theorem proving* is a manual process, similar to mathematical proofs, in which certain assumptions are made regarding the environmental configuration of the system under test. Runtime verification, as it applies during execution has no such limitation. *Testing* shares with runtime verification the title of being an "incomplete verification technique", in that neither claim to consider every possible execution of the system. Additionally, Leucker and Schallhart observe similarities to *oracle-based testing* (in which fixed input sequences are verified) and *passive testing* (in which the input sequences are not predetermined) [20]. Since model checking and theorem proving often refer to an incomplete model of the system under analysis, the use of runtime verification through monitoring provides an attractive mechanism to verify the actual system which was implemented and can be used in combination with other approaches. Our approach bridges testing and runtime verification, rather than formal methods such as model checking or theorem proving, and makes use of contracts, extending on the syntax of EAGLE (with light-weight properties of LTL), to do inline monitoring (at runtime rather than compile time) to verify security vulnerabilities in C-based Linux binaries [43, 20, 41].

A more recent systematic literature review was conducted by Rabiser *et al.*, building upon earlier taxonomy work of Delgado *et al.* [51] and Dwyer *et al.* [52], in which they derived a comparison framework for runtime monitoring approaches applied to 32 different implementations [21]. Backing their systematic literature review, 2,201 papers were identified as potentially relevant (published between 1994 and 2014). This set was reduced to 1,235 papers that were considered unique and in scope based upon their abstracts, then further reduced to 365 papers being in-scope based upon specific criteria (e.g. must perform continuous runtime monitoring of requirements [offline excluded], be peer reviewed, and be available for download in English). From the remaining 365 papers, 65 were ultimately selected after further analysis which covered 50 distinct techniques. Publications were only excluded if all authors agreed. In order to classify 50 approaches which were covered by the 65 papers, Rabiser *et al.* then defined, and refined, their classification framework to cover 4 top-level dimensions {*context, user, content, and validation*} and 21 elements which are distributed across dimensions (7 mandatory {*goal and scope, approach inputs, approach outputs, language, mapping to underlying technology, constraint patterns, and nature of validation*}). Only 30 approaches (out of 50) provided the mandatory elements, and two additional approaches were added following peer review, resulting in the classification of 32 approaches. While the results of their findings (categorizing 32 distinct approaches) found the majority of approaches were involved in verifying a system against its specification, it also indicated only ConSpec [23, 39], RMTL [24], and Serenity [22] as having capabilities relative to Security monitoring, however, Mobucon-EC has also been evaluated in a case study related to security [25, 53]. According to Rabiser *et al.*, the vast majority of approaches to runtime monitoring require advanced skills such as formal backgrounds (e.g. Event Calculus and LTL), domain specific languages, or writing custom rules/probes. Furthermore, while some approaches reuse existing formalisms and languages (e.g. Event Calculus and Object Constraint Language), roughly one third of the 32 approaches analyzed used their own Domain Specific Language. Constraint patterns, one of the elements used for the content dimension, identifies event sequences, presence/absence of an event, and data/performance properties as primary forms of constraint specification. For security, input validation related vulnerabilities usually simplify into the verification of data properties at specific call sites. A subset of approaches also use data and event manipulation to store, and potentially modify, data from different events. In the case of input validation, the data inspected in a constraint pattern are often recorded

as part of an earlier event. The majority of the approaches covered in the review were not available to the public, nor were they validated beyond simple examples for illustration. Our CB_SAMF, combined with static analysis approaches, is evaluated using two new test suites that we have publicly contributed to the Software Assurance Reference Dataset (SARD) in collaboration with the National Institute of Standards and Technology (NIST). To enable the comparison of our approach to other online runtime verification/monitoring approaches we have populated Table 2.1 according to the four dimensions and 21 elements.

Dimension	**Element (Mandatory *)**	**CB_SAMF**
Content	Specific goal & scope*	Security monitoring for violation of vulnerability predicates
	Life-cycle support	Development, maintenance, and testing
	Application domain(s)	Linux-based binaries (user and kernel)
	Architectural style(s)	Any
	Approach inputs*	Contract
	Approach outputs*	Probes, monitors, as well as security violations and safety/liveness results (deployed probes and monitors)
	Intrusiveness and overhead	Runs within the system
User	Target group	Engineers, security auditors, QA staff
	Motivation	Detect security violations
	Needed skills	Programming skills and Domain Specific Language
	Input guidance	None
	Output guidance	None
Content	Language*	Domain specific language (CB_SAMF contracts)
	Reasoning and checking*	Custom tailored mechanism(s) for evaluating rules and constraints
	Constraint patterns*	Occurrence/ordering events, custom functions, saftey/liveness properties, and data checks/aggregation
	Data and event manipulation	Aggregation and analysis of arbitrary data (e.g. system properties, variables, statistics)
	Trigger	Events
	Meta-information	No
	Variability and evolution	Probes can be (de)activated
Validation	Nature of validation*	Empirical evaluation, examples, and case studies
	Availability and support	N/A; researchers still around

Table 2.1: Categorization of CB_SAMF relative to elements of the runtime monitoring categorization of Rabiser *et al.*

Our approach is similar to the work of Gunadi and Tiu in that both approaches address monitoring in the domain of security, provide instrumentation through kernel modules, and the need to represent *history*. However, their past-time subset of metric temporal logic called RMTL is based upon time intervals, they use a policy approach similar to contracts for specifying constraints, and their prototype targets only privilege escalation vulnerabilities in the Android operating system [24]. Gunadi and Tiu state

that privilege escalation is a difficult control flow problem which would typically be solved using static analysis. Rather than re-solving static analysis problems they developed a causal dependency heuristic to flag a violation if a series of events occurred within a particular time frame. In practice there are close to one thousand unique weakness types that can lead to vulnerabilities, of which privilege escalation is one class (e.g. CWE-250). Our evaluation targets five vulnerability types related to input validation. Gunadi and Tiu's work came the closest to implying that a relationship between static and runtime verification needs to be drawn, however, the distinction was never made (see our contribution in Chapter 3).

Another security related approach is presented by Aktug and Naliuka with their ConSpec policy specification language for specifying user policies and application contracts across the development, deployment, and runtime phases of an SDLC through the definition of automatons [23]. Aktug and Naliuka focus their monitoring approach on the production/maintenance phase of Android mobile devices. Since they are targeting production, a stated goal is to reduce performance overhead by addressing policy concerns by first deploying a static verification model checking approach during development, with the restriction that policies which are not enforced by the contract can be enforced by monitoring at runtime (maintenance phase), which involves instrumenting/refactoring the code with monitoring hooks at installation time of the application (prior to execution). Aktug and Naliuka's approach to using model checking to reduce concerns increases the burden on the developer and incurs a large performance cost which many applications would not warrant. Since we target the development through testing phases, and look to combine static analysis and runtime monitoring for vulnerability detection and verification, the production environment performance concerns related to using runtime monitoring for verification is lessened. Furthermore, Aktug *et al.* further define the monitoring framework for ConSpec as a mechanism that targets the Java virtual machine (JVM) and as such cannot be used for verifying contractual obligations at levels below the JVM which resides in the application layer of the software stack [39]. Our approach is capable of instrumenting code both in application and kernel space.

Khan, Serpanos, and Shrobe present an interesting runtime model-chekcing approach in which they claim to have contributed a rigorous (sound and complete), real-time security monitor for embedded systems [54]. Specifically, they claim to have formally proven the absence of false alarms, however, in general the consensus is that soundness guarantees that there will be no false negatives, and completeness

guarantees no false positives [55]. Furthermore, Khan *et al.* use a model-based system that uses normal/good behavior as a reference and raises alarms when deviations occur (i.e. anomaly-based system). We fail to see how this approach can avoid the typical issues on anomaly-based systems discussed above related to IDSs which include the possibility of both false negatives and false positives. Another challenge for anomaly-based approaches is that there needs to be a validation that the implementation and specification themselves are not flawed. The example given in the paper is a specification for the PID algorithm for a controller that controls the water level of a subsystem that includes a feed-water tank and a pump. For the example, the authors provide a model for the computations performed by the PID controller and a model of the feed-water subsystem. The soundness proof is built upon the presumption that the specification of the model is correct and all-encompassing, before comparing the execution pre/post-state(s) with the specification pre/post-state(s) in order to determine if an alarm should be raised. How can this base assumption that the model is correct be guaranteed in order to support the claims for soundness and completeness? Finally, while Khan *et al.* do conduct a performance evaluation of their approach to show how runtime performance is impacted by their simulated experiment, there is no recording of the amount of development effort, or cost, for creating the models that apparently have to be authored by hand (also introducing the possibility of human error). Human error in the creation of the models will likely violate the base-claim of being sound and complete in all but the simplest of systems. In general it is very difficult to be sound and complete for a small subset of weakness/vulnerability types, let alone "all weakness types". This is true of both runtime verification and static analysis approaches, as can be seen in the new "Ockham" track in this years Static Analysis Tool Exposition (SATE) which has its focus on sound analyzers[5].

The ability to compare vulnerabilities identified by different analysis products is a difficult challenge approached by Bush in [56]. Ultimately, the work of Bush, funded by Department of Homeland Security, is aimed at automatically scoring static analysis tools ability to find memory violation vulnerabilities in six real-world programs used by the Static Analysis Tool Exposition (SATE) that is sponsored by the National Institute for Standards and Technology (NIST). Bush takes a formal approach called *abstract interpolation*, using the CodeHawk abstract interpolation engine, which is a scalable technology for proving properties about programs over all execution paths. Bush states that "[...] it is in principle possible to determine, by formal methods, where

[5]https://samate.nist.gov/SATE.html

such vulnerabilities exist, but as a practical matter, formal methods do not currently scale well to real-world programs [...]". This is evidenced by the stated memory and processing overhead of the approach, combined with the reported requirement to manually create required lemmas for a particular program by hand to cover roughly 33% to 45% of the proofs for each program. Furthermore, while formal methods are not presently scalable, Bush claims that the use of static analysis tools are the first line of defense for identifying large numbers of vulnerabilities that developers miss. While static analysis is not sound (reporting only true vulnerabilities - there are false positives), nor complete (reporting all vulnerabilities - there are false negatives), we can combine the use of static analysis results with runtime monitoring approaches to verify suspected security weaknesses to produce a more sound subset of results.

2.2 Contracts

The notion of using contracts to improve different design processes in software engineering has been proposed and investigated by many researchers [57, 58, 59, 60, 32, 61, 62, 31]. Software Contracts (or Contracts for short) have been proposed for reliability and formal verification; although, their use in security is limited [57, 58, 59, 60, 61, 62, 31].

Older work by Bertrand Meyer [57] on design by contract is also an inspiration for this work. Meyer identifies debugging for reliability as the main application for runtime assertion monitoring. In his article, Meyer discusses the use of three primary contractual components for use in design by contract towards improving reliability in software systems. They are essentially preconditions, postconditions, and invariants. As many readers are familiar with the concept of preconditions stating what must be guaranteed prior to a function, and postconditions as statements that must be guaranteed after a function, we only need to describe what an invariant is used for. Invariants, in general, are features that remain unchanged after operations are performed against the features. Meyer identifies these three types of assertions for providing the basis for a software contract (using the Eiffel language for example):

1. ***Precondition*** - Expresses requirements that any caller must satisfy before calling a routine
2. ***Postcondition*** - Expresses requirements that are ensured after a call to a routine

3. ***Class Invariant*** - Expresses features that remain unchanged after operations are performed against the features (specifically in the context of a class)

Meyer proposed the use of contracts during the design phase in object-oriented systems to improve system reliability [57]. Meyer argued that through the use of "design by contract", developers could improve correctness and robustness of systems leading to the absence of bugs (this proactive approach for removing defects is an inspiration for our work). A typical contract outlines some benefits for each party as well as some specific obligations. An example template of a contract as proposed by Meyer can be seen in Table 2.2. This example depicts one of the fundamental properties of a contract: an obligation for one party is usually a benefit to the other party. Furthermore, one of the interesting conclusions drawn from these contracts by Meyer is that if an assertion is violated during runtime monitoring, each violation of an assertion indicates the existence of a bug. A violated precondition implies a problem with the caller (client), while a violated postcondition implies a problem with the callee (supplier). Finally, at the end of the paper by Meyer, a question remains related to what should happen if one of the contractual parties is not honest in their pledge to fulfill their part of the contract. While this may not be a deal breaking issue with reliability, it most certainly is with security. Our form of contracts, focused on verified security vulnerabilities, adds the expressive notions of *environmental*, *historical*, and *reactive* clauses in order to both capture and respond to security vulnerabilities. Normal preconditions, postconditions, and invariants are not able to express contextual requirements, requirements which may have been violated/satisfied previously, nor specify how a violation should be handled. An example of such a situation that must be handled for security is related to taint/dataflow analysis of input violations which is typically found by static analysis. The source location of taint analysis is indicating where untrusted input is entering the dataflow, while the sink is specifying where something undesirable could occur if tainted data reaches it. It is only when a subset of input enters the system and reaches the sink that something undesirable does typically occur. A precondition for the sink function above does not have the ability to understand what the postcondition of a function several calls before it satisfied (nor what properties the resulting data had).

McKim makes the point that programming using contracts with languages other than Eiffel is possible and we may also desire more contract features than those provided

Party	**Obligations**	**Benefits**
Client	Provide *XYZ* in order to obtain *ABC*.	ABC
Supplier	Consume *XYZ* for providing *ABC*.	XYZ

Table 2.2: A contract typically has at least two parties (a supplier and a client/consumer). An obligation of one party is often the benefit of the other party.

by Eiffel [58]. When McKim wrote the article, Eiffel was the only commercial language supporting contracts, even though languages like C and C++ provided underlying support for the concept of contracts using an assert macro. McKim concludes that contracts provide a useful guarantee to other vendors consuming their components. Specifically, that the component has been validated against its specification.

Leslie Lamport also approached the use of contracts when deriving an approach for specifying concurrent systems [59]. Lamport's work provided a foundation for formal validation of systems based on their specifications and state transitions. With the ability to handle linear temporal logic statements, safety properties, and liveness properties there were multiple similarities to the more recent work of Barringer *et al.* [44, 43]. Later Lamport continued this work with Abadi; they approached the problem of composing specifications from a purely semantic point of view [63].

Février *et al.* proposed the use of contracts for specifying the interactions between objects using either message passing or flows. Their framework has many similarities to the work of Lamport [60]. Their realization of contracts is a process that observes and arbitrates the collective behaviour of configurations of objects. Février *et al.* point out that the contract description enhances the reusability of components since it goes beyond the syntactic definition of their methods. As with most of these approaches, Février *et al.* is limited to providing a framework that produces log events and does not support specification of additional information that may be required to identify security contracts. Finally, we found the notation of the Eagle framework of Barringer *et al.* was more straightforward than the notation used by Février *et al.*, which stands in contrast to their claim that their notation is more approachable to programmers and software engineers. We should attempt to remove most of the overhead of the contract notation from our consumers by trying to automate as much of the assertion monitoring framework as possible.

Renewed interest in using the notion of contracts for the design of complex systems has resurfaced [62, 31]. Cimatti *et al.* base their work on *guarantees* regarding the input/output behavior of a component, provided that the environment obeys certain *assumptions* with the ultimate goal of allowing compositional reasoning, stepwise

refinement, and principled reuse of components for contract-based design [62]. In essence their OCRA tool for verifying contract refinement builds upon the contract-based design of Meyer [57]. In the case of OCRA, contracts are created and provided with components prior to their use. Application security vulnerability verification through runtime monitoring using contracts is fundamentally different in that it is looking to confirm the presence of flaws that violate a contract. Cimatti *et al.*, present a "fully formal" contract framework supporting contract decomposition of complex systems [62].

In addition to using contracts for design, which is similar to the early work of Meyer except the focus is on guarantees and assumptions rather than pre/post/invariant conditions and Meyer's contracts were enforced at runtime [57], we believe that contracts can also be very beneficial for verifying security assertions regarding existing systems.

Finally, several approaches in the area of runtime verification use the notion of policy rather than the notion of contract [23, 24]. The difference between policy and contract can be found by answering the question of enforceability[6]. In business terms, a policy is only enforceable if it is included in the terms of a contract. While policies specified in other approaches to runtime verification share similarities for specifying constraints that cannot be violated in an LTL form, they lack the remediation element which permits for enforcement or actions to be taken when a contract is violated. Further discussion on contracts will be deferred until the model section.

We are beginning to see a focus, both in research and industry, on reducing security vulnerabilities during system development [2, 1, 4]. A monitoring framework that can be used throughout the SDLC, based on lessons learned from IDSs, and using contracts enforced during runtime would provide a mechanism to assist in the reduction of security vulnerabilities. In order to improve vulnerability detection and mitigation we must have suitable measures and metrics to identify and track progress. In the following section we will review existing measurement strategies and approaches.

2.3 Measurement and Metrics

The need for measurement in software is rooted in the notion of quality and we desire security assurance products to not only have quality but also enable companies to

[6]https://www.americanbar.org/newsletter/publications/law_trends_news_practice_area_e_newsletter_home/0705_litigation_employmentcontracts.html

improve the quality of security. Paul Oman and Shari Lawrence Pfleeger combine a collection of papers on applying software metrics in [64] which pre-date most security assurance metrics. Measurement allows for the quantification of concepts or attributes in order to manipulate and learn more about them. When measuring an entity we are in fact creating a mapping from the empirical world to the mathematical one.

The concept of metrics is analogous to the practice of scientific measurement that is used to indicate progress or achievement. We need to think carefully on how to apply measurement strategies to the issue of system security. Existing metrics found in other areas of science and engineering have the potential to be used for measuring security properties; however, most of them are not a natural fit for security assurance. Existing metrics that may be relevant include the following:

- Changed Source Instructions (CSI) [65]
- Cyclomatic Complexity (CC) [67]
- Defect Density [65, 67]
- Defects/KLOC [67]
- Lines of Code (LOC) [65, 1, 67]
- Mean Time Between Failures (MTBF) [66, 67]
- Mean Time To Failures (MTTF) [65, 67]
- Number of Security Breaches [1]
- Number of Security Breaches/Lines of Code [1]
- Security Defect Density [1]

Considering the various approaches to security analysis and evaluation that are in use today, several metrics are inappropriate. For example, computing the mean-time-to-intrusion (MTTI), which is similar to mean-time-to-failure (MTTF), for a vulnerability that is already known to exist and be exploitable is a waste of resources. If we know an exploitable vulnerability exists we should put our resources toward removing it. In this section we will briefly discuss the history of metrics used for security before identifying metrics that exist today, which may be applicable to our empirical evaluations, and specify areas that warrant additional metrics.

2.3.1 History of Metrics in Security

Metrics are measurements that are intended to span a temporal duration and be SMART (specific, measurable, attainable, repeatable, and time-dependent) [68]. The number of proposed security measures and metrics in the literature is now in the hundreds and sufficient to warrant the creation of assessment frameworks helping to determine which metrics are suitable for a particular Information Security Management System (ISMS) [69]. Heinzle and Furnell, for example, created a prototype cataloging 95 metrics, gathered from eight sources, to associate metrics with security requirements and ISO 27001 clauses and controls.

Many measures are poorly defined and not rigorously validated [10]. Furthermore, some software metrics such as code complexity [70], as shown by Shin and Williams, appear to be a poor indicator for the presence of vulnerabilities in code [71]. Vulnerability prediction metrics are studied by Shin and Williams with the goal of predicting where vulnerabilities may occur. The assertion that faults and failures being correlated to complexity apparently does not always hold for vulnerabilities. While Shin and Williams experiment was limited to nine complexity metrics, applied against the JavaScript engine in the Mozilla application framework (written in C/C++), the results indicate that vulnerability prediction via complexity metrics can be achieved with low false positive rates and high false negative rates. The high false negative rates appear to contradict the work of McGraw [1]. Furthermore, De Oliveira *et al.* state that "[c]ode complexity is a feature of the programming language that, in theory, has no impact on whether there is a vulnerability or not" [72]. The primary goal for security should be low false negative rates with a reasonable level of false positives and thus we should question any implied correlation between code complexity and vulnerabilities. Using test suites to evaluate security assurance tools ability to detect vulnerabilities using various coding constructs that add complexity, as discussed by De Oliveira *et al*, provides evidence that tools are able to detect vulnerabilities under these conditions. Evidence does not appear to support that the presence of code complexity implies a higher likelihood of vulnerabilities occurring.

Lipner asserts that open source is not inherently more secure than closed [73]. Additional metrics indicate, however, that code review can improve security, provided that the reviewer has security experience and collaborator familiarity. Meneely, *et al.*, empirically studied whether vulnerabilities are captured by *Linus' Law* (term coined by Eric Raymond) of "many eyes make bugs shallow" for code review in the context of open source software development of the Chromium Browser project. Due to vulnerability-related socio-technical familiarity deficiencies of developers conducting code review, and the experience of the developers, vulnerabilities are often missed during code review even when the number of reviewers and number of reviews is higher for files containing vulnerabilities [74]. The results of the experiment by Meneely *et al.* is limited to only one project and the results may not scale, however, it does indicate the necessity of compensating for developers lack of experience with secure development practices and software security.

Pendleton, *et al.*, created a logical grouping of metrics into a framework based upon understanding attack-defense interactions [75]. The four metric sub-groupings

include those related to (1) system vulnerabilities (V), (2) defense power (D), (3) attack or threat severity (A), and (4) situations. These metrics are combined into a single formula to represent the result of an attack-defence situation at time *(t)*: $situation(t) = f(V(t), D(t), A(t))$. Related to our work, is the specification of a vector *V(t)* representing an enumeration of vulnerabilities that exist despite patching and defense on a given system. Vulnerability metrics are divided across three sub-domains related to *user* (e.g. social-engineering related), *interface-induced* (e.g. attack surface), and *software*. Software vulnerability metrics are then refined into three logical groupings related to the following: *temporal attributes*, *individual*, and *collective*. Metrics for temporal attributes of vulnerabilities are intended to measure the evolution of vulnerabilities (e.g. how many detected/exploited vulnerabilities have/will occurr(ed) in the past/future) and vulnerability lifetime (e.g. how long it takes to patch a vulnerability once disclosed). Metrics for individual software vulnerabilities focus on measuring the severity of a particular vulnerability (e.g. Common Vulnerability Scoring System (CVSS)) or category of vulnerabilities (e.g. Commmon Weakness Scoring System (CWSS)). Metrics related to measuring the severity of a collection of vulnerabilities focus on deterministic and probabilistic attack graphs. While the distinction that each computing system has an associated set of vulnerabilities, a subset specific to software vulnerabilities, the definition does not go further to identify that each software program on a computing system has a set of distinct vulnerabilities based upon underlying weaknesses. Furthermore, the survey by Pendleton *et al.* does not cover metrics related to measuring the efficacy or efficiency of products/tools for detecting/protecting against vulnerabilities/attacks and their underlying weaknesses. Finally, related to defense metrics, Pendleton *et al.* also identify four metrics related to monitoring. Directed toward intrusion detection system monitors (IDS) the following measurements are still relevant: *coverage*, *redundancy*, *confidence*, and *cost*.

Finally, Bird summarizes *technical*, *operational*, and *executive* metrics targeted to measuring the effectiveness of an AppSec program [76]. Specific to vulnerabilities, Bird identifies the following technical measurements that should be tracked as input to metrics: *Number of Vulnerabilities Found*, *Vulnerability Density* (e.g. vulns/KLOC), *Severity*, *Vulnerability Type*, *Method of Discovery*, *Number Fixed*, and *Time-to-Repair*.

2.3.2 Applicable AppSec Metrics

Metrics and metrology (scientific study of measurement) for security assurance products/tools is an active area of research that is important to industry, academia, and government [77, 10]. Verendel specifically states that "[q]uantified security is [...] a weak hypothesis because a lack of validation and comparison between such methods against empirical data" [78].

Recently the National Institute of Standards and Technology (NIST) produced an interagency report which indicated that "most existing measures are only moderately predictive of the high-level properties we wish to determine" and that there is "not even extensive and detailed data, such as numbers and types of vulnerabilities, upon which measurement research might be based" [10]. Black *et al.* further identify three areas of measurement that require attention: (1) encouraging the use of measures, (2) process measures, and (3) measures of software as a product. Software measures are then broken down according to four dimensions towards classifying measures within a taxonomy: (1) how "high-level" is the measure, (2) static/source or dynamic/execution related, (3) point of view (black-box/functional or white-box/structural), and (4) object of measure (weaknesses in code, quality of code, conformance to specification). With measures grouped and applied, Black *et al.* then propose the use of a software security assurance formula backed by the measures. The formula has three inputs relating to the following: assurance from development process (p), static/dynamic analysis assurance (s), and resilient execution environment assurance (e). This formula can provide a measurement that allows for compensation in one area by another. For example, if there is low assurance for the development process, compensation can come from heightened assurance from a resilient execution environment or increased security analysis practices against the developed software artifacts. In order for improvements in assurance to be measured we require suitable measurements and metrics to evaluate a given technique over time during an iterative development model. The five areas identified by the report are marked as targets for improvement over the next three to seven years and require verifiable measurements to make a difference.

In the area of static analysis several metrics have already been proposed in the literature. These metrics include measuring *vulnerability density*, comparing projects by *vulnerability severities*, breaking down results by *vulnerability category*, and monitoring trends such as *vulnerability dwell time* [79]. Furthermore, Okun *et al.* claim that solely measuring the number of weaknesses is not satisfactory due to the fact that not all

weaknesses can result in failure [80]. Thus, they proposed measuring the effect of tool use on the number of reported vulnerabilities per project in the National Vulnerability Database (NVD). Two projects were analyzed (MySQL and Samba), in the context of two static analyzers (Coverity and Klockwork), and then compared against random samples of projects from the NVD. As the authors correctly point out, there are numerous deficiencies in the reliability and accuracy of this metric that reduces its usefulness to a rough indicator that the use of static analysis tools impact CVEs. A side-effect of the work of Okun *et al* is that it adds further weight to the assertions presented in the work of Meneely *et al.*, regarding developers lacking sufficient security expertise while conducting code reviews in open source projects. There is need for more accurate ways to measure the impact of security assurance tools.

Challenges of measuring efficacy of a particular security assurance product, such as static or dynamic analysis tools, are compounded by the difficulty presently facing efforts to compare the results of different tools. As described by Black [81] and Bush [56] these challenges are multifaceted. First, metadata associated with reported vulnerabilities makes it difficult to compare across products because there is no true standard for information such as program locations and constructs [56]. Bush also notes that informal weakness taxonomies, such as the CWE, specify weakness categories that are not disjoint, leading to many-to-many and one-to-many mappings which make it almost impossible to count the number of independent vulnerabilities. Black discusses the challenge of measuring benchmark results due to the presence of potential chains (two or more tightly related weaknesses leading to a vulnerability), composites (two or more weaknesses leading to a vulnerability - not disjoint), or hierarchies (difficulties due to the design of hierarchical taxonomies of weaknesses where tools may report higher-level abstract weaknesses compared to more specific refined types) of weaknesses present in test cases. Both Black and Bush enumerate additional challenges related to accurately measuring results and comparing security assurance products, including counting weaknesses and locating weaknesses in the code. While both papers discuss numerous challenges of current benchmarking and empirical evaluation attempts, proposed improvements to the test suites and the metadata related to the test cases are not presented. Furthermore, these same challenges persist for empirically evaluating runtime approaches to vulnerability detection.

Certain probabilistic metrics appear in almost all detection approaches including IDS, static analysis, dynamic analysis, and even binary logistic regression analysis. True-positive rate (or recall), false-positive rate, true-negative rate, false-negative rate,

and precision are examples of common probabilistic metrics typically employed for evaluation or to compare and contrast two or more approaches [75, 71, 82, 83, 84].

The Juliet test suite of small synthetic programs covering 181 CWEs, created by the NSA, has been used to conduct studies intended to measure specific capabilities of static analysis tools by measuring true-positive, false-positive, false-negative, and true-negative reporting. Common metrics used by Goseva-Popstojanova and Perhinschi for the comparison between tools include the following: accuracy, recall, probability of false alarm, and G-Score [85]. Okun *et al.* discuss the use of false-positive rate, precision (true-positive rate), recall, and discrimination-rate [86]. Many of these studies show deficiencies in static analysis tools for both true positive and false positive rates which raises the question regarding how to measurably improve results for a static analysis tool in order to have a more efficient and mature security assurance process integrated into an SDLC.

Another pair of test suites called 45 and 46, provided by the Software Assurance Reference Dataset (SARD) project at the National Institute of Standards and Technology (NIST), are evaluated by Diaz and Bermejo in [87]. The metrics used for comparison include the following: precision, recall, and F-Measure. The work of Delaitre *et al.* expresses the difficulties in measuring and comparing the effectiveness of bug finding tools relating to metrics such as recall, coverage, precision, discrimination, F-Score, and overlap. Specifically, not all metrics are applicable to all categories of testing artifacts (e.g. production software, software with known CVEs, and synthetic test cases).

Presently, while there is a body of literature beginning to accumulate for runtime monitoring for security vulnerability detection, there is a significant lack of empirical evaluation for runtime monitoring in this area. Halfond and Orso combine static analysis and runtime monitoring to detect a single vulnerability category (SQL injection) and evaluate the approach using two programs [50]. The initial results of their experiment included a false positive rate of 0% and a true-positive rate of 100% using hand crafted inputs (17 legitimate accesses and 10 attacks). While the coverage of the experiment is very limited, it is one of the only examples of research in this area with empirical data to support it. Yang and Zulkernine present a monitoring approach using aspect-oriented wrappers and contracts to detect SQL Injection, Cross-Site Scripting, and access control policy violations in Java applications [88]. The single experiment consists of a single web application (WebGoat) that is used for a case study, however, results are not presented. Recently, Antunes and Vieira conducted

an experiment comparing static analysis, dynamic analysis, and runtime anomaly detection efficacy for detecting SQL injection vulnerabilities in SOAP web services [89]. By measuring true positives, false positives, and false negatives they were able to report precision, recall, and F-Measure results for each approach relative to two benchmarks which included both application source and workloads. Finally, the work done by Jimenez *et al.* supports our belief that there is also a need to customize the focus of vulnerability measurements on a project specific basis [90].

2.3.3 Metrics Summary

Software reliability is measured as the probability that the program works without failure for a specified length of time and is a statement regarding the future behavior of the software [66]. Software security assurance, however, provides a level of confidence related to evidence indicating that a program can execute free of exploitable vulnerabilities.

Many measures and metrics for software related security, such as defect density and Mean-Time-To-Intrusion (MTTI), provide indications of the level of security "after" software has been in execution for a period of time. Other approaches for measuring security assurance tools efficacy, such as precision and recall, can be applied to suitably verified test suites in order to compare approaches or measure a specific tool's ability to detect weaknesses before execution. While test suites have been created for the purpose of evaluating static analysis tools [91, 77, 72], and very recent attempts to create test suites also attempt to address dynamic as well as static [92], we are unaware of work being done for empirical evaluation for runtime approaches for security assurance.

Furthermore, many existing metrics focus on the results of tool investigation rather than improvements to the measurements themselves [87, 93, 94]. Further work that enables stakeholders to measurably improve the efficacy of security assurance tools over time is also warranted. While the debate between formal runtime methods, informal runtime methods, static analysis, dynamic analysis continues, we need improved measurements for *coverage* and *improvement* over time.

We desire detection mechanisms, measurements, and metrics that focus on minimizing false negatives, improve software security assurance practices over time, and leverage security expertise through tools to help address socio-technical[7] shortcomings

[7]A term, from sociology, that is used by Meneely *et al.* to describe the connection between two

of developers familiarity with vulnerabilities. Detection mechanisms should provide meaningful vulnerability metadata to support measurements. Measurements should capture the relevant data associated with vulnerabilities (e.g. vulnerability category, file/line number, vulnerability evidence, and number of Lines of Code (LOC)). Vulnerability metrics should provide actionable data which expose the efficacy and efficiencies of approaches and identify areas for improvement. In the next subsection we will briefly discuss some of the modeling tools available for use with the SDLC and review some of the newer techniques that are used in the security arena.

2.4 Security Analysis Tools

Over the years multiple approaches for analyzing security risk of applications have taken form. These approaches typically fall into three buckets. First, white-box approaches, where complete access to source code artifacts are available, create a model of the target and assess it for weaknesses (e.g. static analysis). Second, black-box approaches, where no access to source code artifacts are available, send data to the input interfaces of the target and measure the responses to look for vulnerabilities (e.g. certain dynamic analysis and fuzzing products). Finally, grey-box approaches where white and black approaches are combined (sometimes called hybrid approaches).

2.4.1 Static Analysis

As a white-box method for software evaluation, static analysis has complete access to the source code of the software under investigation. Essential to quality results from a static analysis product, is a firm understanding of the cause and effect of the static analyzer being studied. While the ruleset for the static analyzer is responsible for clearly defining what is being looked for, the translators must create an accurate model of the program being studied and ensure the analyzers effectively and efficiently detect what the ruleset specified. It is up to the individual running the static analysis product, however, to ensure that all necessary source code is provided to the analyzers to ensure coverage. There are multiple approaches used in static analysis. For example, HP Fortify SCA static code analyzer supports the following analyzers [79]:

---

people in which both social and technical aspects are found in their work-related collaborations [74].

Buffer: Detects boundary violations of declared memory buffers.
Control-flow: Detects potentially dangerous sequences of operations.
Dataflow: Detects potentially dangerous, inter-procedural, use of tainted data.
Semantic: Detects potentially dangerous functions at the intra-procedural level.
Structural: Detects potentially dangerous flaws in the structure of the program.
Configuration: Detects mistakes, weaknesses, and policy violations in an application's configuration files.

Static analysis has been shown in some controlled experiments to locate vulnerabilities faster than a targeted penetration testing approach [95]. It has also been shown that using static analysis approaches can lead to relatively high rates of false positives [87]. While static analysis can detect potential vulnerabilities in a timely fashion, when the set of suspected vulnerabilities grows, it becomes increasingly important to have a strong confidence that they are in fact exploitable vulnerabilities (true positives).

2.4.2 Runtime Monitoring

In contrast with dynamic security scanners, runtime monitoring is not a black-box approach, nor is it a white-box approach to software evaluation. Runtime monitoring has access to the properties of the system while it executes. A common example of a technology that can be used for runtime monitoring of a system while it executes is a software debugger (e.g. the GNU debugger GDB). Such runtime monitoring systems provide access to a wide variety of software system attributes, which include the following:

- View/modify values of variables
- View/modify values of registers
- View/modify values of memory
- Set/delete/enable/disable breakpoints
- Step through assembly instructions
- Step through source instructions
- Continue executing to completion (or next breakpoint)
- Debug running software by attaching to it during runtime.

From practical experience, it is possible to conduct monitoring of software without the inclusion of symbolic information, however, it hinders most of the more useful features listed above (such as stack traces with symbolic names of functions). This is also true of CB_SAMF since symbolic information is used by the predicates to target

the probe insertion points and to collect runtime information. The target usage of this novel approach is during development and testing of software rather than on production systems. Ultimately, to improve the state of software security we need to make greater strides to help the community release software that has fewer weaknesses in order to reduce the attack surface. As a result, the cost of exploiting systems increases and the risk to companies decrease.

2.4.3 Current State of Static and Dynamic Approaches

In 2014 the Institute for Defense Analysis (IDA) published the "State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation" report [5]. While the stated goal of the report is to assist Department of Defense (DoD) staff make effective software assurance decisions, it also provides categorization of tools and techniques for identifying security vulnerabilities. Larsen *et al.*, describe their approach to measuring the abilities of different vulnerability detection tools and techniques as primarily a subjective/qualitative approach involving examination of collected information by subject matter experts, debate of findings, and agreement between authors [5]. Furthermore, they also note that there are efforts by NIST Software Assurance Metrics and Tool Evaluation (SAMATE) and NSA Center for Assured Software (CAS) which are working to develop test suites to evaluate a few specific tool/technique types, however, there is still a great deal of work remaining for the community to effectively measure these tools and techniques. Larsen divides static analysis into 15 different tool or technique groups and separates dynamic analysis into 19 different groups. Within static analysis, in relation to "source code analyzers", Larsen divides tools into six groups: 1) Warning flags, 2) Source code quality analyzer, 3) Source code weakness analyzer, 4) Context-configured source code weakness analyzer, 5) Souce code knowledge extractor for architectural/design/mission layer information, and 6) Requirements-configured source code knowledge extractor. We are primarily interested in only (3) and (4) for the identification of security weaknesses. In theory, any source code weakness analyzer or context-configured source code weakness analyzer could be the source of identified potential vulnerabilities which CB_SAMF could be used to perform runtime verification. While the input to static analyzers is typically source code, the input to CB_SAMF is contracts which operate against binaries. Source code weakness analyzers are what most people think of when discussing "source code security analyzers" or "static application security testing",

where source code is parsed and translated into a model of the program to which weakness-identifying rules can be applied during a scanning phase. Context-configured source code weakness analyzers are simply those analyzers that are capable of having custom rules added to their arsenal to identify weaknesses specific to code being evaluated (e.g. rules to cover third-party libraries and frameworks used by the code being evaluated). Related to these two groups of tools, Larsen *et al.* also make several additional distinctions:

1. ***Input format***: Source code, bytecode, and binary variants of weakness analyzers. Distinction is made based upon the input file format that the model is created from.
2. ***Context-configured***: Allowing for the distinction that additional rules can be written to improve accuracy and coverage for product specific analysis. The primary observable side-effect is the increased expectation on subject matter expert (SME) experience required to write the custom rules.

Within the 19 groupings for dynamic analysis, *host-based IDS/IPS/Integrity checker*, *automated monitored execution*, and *debugger* are the most related to our interests. Host-based IDS/IPS/Integrity checkers monitor data, other than network traffic, for malicious activity. Automated monitored execution isolates a program, executes it, detects malicious activity and reports findings prior to installation. Finally, a debugger enables observation and control of a program during execution [5]. In addition to static and dynamic approaches, Larsen *et al.* also describe hybrid analysis approaches to vulnerability analysis where static and dynamic approaches are tightly integrated. None of the listed approaches, however, combine static analysis for identification of potential weaknesses with dynamic analysis for verification of reachability/exploitablilty. We believe that our proposed process, model, and evaluation is the first to combine static analysis weakness identification and runtime monitoring for verification. Interestingly, in Section 9 of [5], 23 gaps in vulnerability analysis were observed. Integration of different results, such as those from static and dynamic, is noted as a difficult problem requiring further work. Obtaining quantitative data on tools and techniques is also lacking, especially related to obtaining "ground truths". The last gap we find particularly interesting is that many tools do not focus enough on reducing false negatives, partly because many companies focus on reducing false positives.

There are several tools already available in the marketplace for scanning source code for possible security violations. The majority of these tools are deemed to be static source code analysis tools and the early tools, listed in Table 2.3 along with a URL, provide a brief description, and targeted language listing.

We will now describe the basic capabilities of an example of more mature tools.

2.4.4 HP Fortify SCA

The Source Code Analysis (SCA) suite put forth by HP Fortify is a mature commercial product that assists companies in securing their source code. One of the most significant contribution this tool set provides is a robust rule building framework that places it in the classification of both the source code weakness analyzer and context-configured source code weakness analyzer groupings mentioned above. The tool can be compared against existing tools and literature related to misuse cases [96, 97].

Using this tool, a developer or security expert can scan existing source code for known risks (specified by the tool using the rule sets identified above). What makes this tool powerful is its ability to expand and add new rules to the environment as they are detected. SCA comes with a mature graphical user interface (GUI) that allows for interaction with existing integrated development environments (IDEs) and a separate tool for reviewing generated reports. One disadvantage of this product, and the majority of static analysis products, is that it cannot detect some types of security risks due to the fact that static analysis is only capable of finding known patterns in the model which is generated from the input source code. Our CB_SAMF can also detect known patterns, however, because it evaluates the system during execution it can also detect patterns which are unavailable in a static model. In addition, statically identified weaknesses have the potential of being false-positives (depending on the type and analysis technique). Another advantage, however, is that the product allows for the graphical representation of the series of function calls, using sequence diagrams, that led to the risk. Lastly, the product also provides details on each type of risk so that a consumer of the product is not only able to scan and identify potential risks, he/she is also provided with a description of what caused the risk and recommendations of how to mitigate it.

NAME	URL	DESCRIPTION	LANGUAGES
BOON	http://www.cs.berkeley.edu/ daw/boon/	Detect buffer overrun vulnerabilities	C
BOOP Toolkit	http://boop.sourceforge.net/	Program model checking	C
BLAST	http://mtc.epfl.ch/software-tools/blast/	Program model checking	C
C++ Test	http://www.parasoft.com	Static analysis and unit testing	C++
CQUAL	http://www.cs.umd.edu/ jfoster/cqual/	Check properties of programs	C
ESCJava2	http://secure.ucd.ie/products/opensource/ESCJava2/	Detect runtime errors	Java
Flawfinder	http://www.dwheeler.com/flawfinder/	Detect security vulnerabilities	C, C++
JCAVE	http://www.sics.se/fdt/projects/vericode/jcave.html	JavaCard Applet Verification	Java
ITS4	http://www.cigital.com/its4/	Detect security vulnerabilities	C, C++
MC	http://metacomp.stanford.edu/	Build compiler extensions (GCC)	C
MOPED	http://www.fmi.uni-stuttgart.de/szs/tools/moped/	Model Checker	Pushdown-systems
MOPS	http://www.cs.berkeley.edu/ daw/mops/	Detect security vulnerabilities	C
PMD	http://pmd.sourceforge.net/	Scans source code for problems	Java
PScan	http://nixbit.com/cat/system/networking/pscan/	Scan for printf-related vulnerabilities	C
RATS	http://www.fortifysoftware.com/security-resources/rats.jsp	Detect security vulnerabilities	C, C++, Perl, PHP, Python
SCAC	http://www.fortifysoftware.com/	Detect security vulnerabilities	C, C++, Java, .Net, etc.
Splint	http://splint.org/	Detect security vulnerabilities	C
Static Driver Verifier	http://www.microsoft.com/whdc/devtools/tools/sdv.mspx	Detect flaws in drivers	C
Uno	http://spinroot.com/uno/	Simple security static analysis	C

Table 2.3: Automated source code security vulnerability scanners.

2.5 Summary

In summary, intrusion detection systems provide a necessary approach to dealing with existing software that misbehaves at runtime. Anomaly-based detection looks for unusual states, signature-based detection looks for states known to be bad, and specification-based detection looks for states known not to be good [3]. Unfortunately, IDSs and IPSs are focusing on detecting and patching security problems when they occur in production systems rather than preventing them in the first place. Since intrusion detection is focused on the maintenance phase of the SDLC, our approach attempts to detect and assist in the resolution of security vulnerabilities during the development and testing phases. This approach is intended to help veto the fact that security exploitations are primarily caused by inherent weaknesses written into the software, either intentionally or unintentionally, leading to security vulnerabilities. None of the above approaches to contracts handles security requirements specifically. We propose a form of contract for our monitoring framework that is used to specify the environmental and system security conditions necessary for the generation of probes which will monitor security assertions during runtime. The violation of a contract during runtime will allow us to identify exploitable security vulnerabilities and our framework will provide mechanisms allowing for reactive countermeasures. Finally, our monitoring framework will also provide an avenue for validating if an identified vulnerability, such as those detected by static analysis products, is actually reachable and exploitable.

Chapter 3

Weakness Identification and Vulnerability Verification

In the next decade there'll be a shortage of great software engineers. We'll be scouring the schools for them.
-Bill Gates

Vulnerabilities are the result of security or quality flaws that have been introduced into the code base during implementation, potentially by architectural oversight, poor quality control or deficiencies in requirements, which violate security policies. As a security assurance community we must continue to improve our ability to identify, verify, and fix software weaknesses as part of our software security assurance measures.

Security vulnerabilities, and their underlying weaknesses, can be identified using a variety of methods including static analysis, dynamic analysis, penetration testing, fuzzing, and code review/inspection. Each of these approaches has the possibility of reporting both false positives and false negatives, however, the meaning of these terms depends on the approach. A false positive in dynamic analysis is referring to a vulnerability, however, static analysis of source code is actually referring to a weakness. In this chapter we clearly define the terminology to be used for vulnerability verification, introduce our general methodology for weakness identification and vulnerability verification, discuss how our methodology can be integrated within the SDLC, and describe the five primary weaknesses used during the remainder of this thesis.

The following sections will introduce the terminology and our general methodology

for the identification of possible weaknesses, potential verification of vulnerabilities, and remediation of the underlying defects.

3.1 Terminology

Existing literature often uses terms interchangeably, however, we provide in this section a collection of terms that should be used specifically to improve clarity of discussion of what is identified, verified, and resolved when trying to improve software security.

3.1.1 Specific Diction in Application Security

Standard definitions from the online Oxford English Dictionary provide us with the following cornerstone terms related to our discussion of application security.

Issue: An issue is "A matter which remains to be decided; a significant matter for debate or discussion" [98]

Defect: A defect is "A shortcoming or failing; a fault, blemish, flaw, imperfection" [99] and is synonymous with fault or flaw

Weakness: A weakness is "[t]he quality or condition of being weak, in any sense of the adj.; deficiency of strength, power, or force" [100] such as "[w]anting in material strength, unsound, insecure" [101] and may be considered synonymous to a *fault* or *defect*.

Vulnerability: A vulnerability is "[t]he quality or state of being vulnerable, in various senses" [102] such as being "[o]pen to attack or assault by armed forces; liable to be taken or entered in this way" [103]

Exploit: An exploit is "[a] means or method of taking advantage of a flaw or vulnerability in software or hardware, typically for malicious purposes (such as installing malware or gaining remote control of a system)" [104]

Attack: An attack is "[a]n attempt to disrupt a computer system, network, etc., by gaining unauthorized access or control" [105]

Alternatively, NIST provides the following definitions:

Weakness: A *weakness* is a defect in a system that may (or may not) lead to a vulnerability [91].

> ***Vulnerability:*** A *vulnerability* is a property of system requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited resulting in a security failure [106].

These definitions for weakness, vulnerability, and attack are in keeping with other attempts to provide concise meaning to application security terminology, such as version 2 of the Internet Security Glossary (RFC 4949) produced by the IETF [107].

Contextually, a weakness is a *defect*, flaw, or fault in the implementation or design which causes the system to be potentially insecure or vulnerable. We use the term *issue* to refer to a suspected *weakness* which remains undecided as to whether or not it is a *vulnerability*. The presence of one or more weaknesses, that are reachable and exploitable, causes a system to be vulnerable to *attack*. An attack on a computer system occurs when an *exploit* is leveraged to take advantage of one or more vulnerabilities resulting from underlying weaknesses or flaws.

Finally, we can connect the above definitions by summarizing that a successful attack is the realization of a perceived threat, which depends on the existence of one or more vulnerabilities in the target, which in turn often depends upon the presence of underlying flaws or weaknesses in the design, code, or configuration of the target.

3.1.2 Weaknesses

Weaknesses have been defined and logically grouped in several different ways by several organizations. MITRE created the Common Weakness Enumeration (CWE) as a hierarchical formal list of weakness types[1]. Another alternative, created by Tsipenyuk, Chess, and McGraw, is the "7 Pernicious Kingdoms" (7PK) collection of software security errors[2] [108].

Within the CWE hierarchy there is a class of weaknesses known as "Improper Input Validation" (CWE-20), which roughly aligns with the "Input Validation and Representation" kingdom from the 7PK, describing a group of weaknesses related to improper input validation or representation related to metacharacters, alternate encodings, and numeric representations. Weaknesses within this logical grouping, which will be used for experimentation in this dissertation, that are both highly likely and result in high impact, include the following:

[1] https://cwe.mitre.org
[2] https://vulncat.fortify.com

Format String: An attacker who can control a function's format string argument is able to cause exploitations such as buffer overflows.

Path Manipulation/Resource Injection: Consuming user input and allowing it to control paths used in filesystem operations can enable the attacker to modify or access protected system resources.

Command Injection: By not specifying an absolute path, a developer is opening the door to potential execution of malicious binaries via the manipulation of the program's execution environment (e.g. PATH environment variable).

SQL Injection: Attacker controlled user input that is consumed as part of a dynamic SQL statement can result in unintentional SQL query execution.

Cross-Site Scripting: Sending un-validated data to a web browser can result in the execution of malicious code by the browser.

3.2 Verification of Weaknesses

Our general approach for identifying weaknesses and verifying they are vulnerabilities, that may be exploited (depicted in Figure 3.1), can involve the application of several analysis and verification approaches such as static analysis, dynamic analysis, penetration testing, fuzzing, and code review/inspection. Static analysis is conducted on some version of the source code (e.g. source, object, or binary) without actually executing the program. Dynamic analysis is conducted against a program under execution. Penetration testing has a much broader scope and can employ the use of static (where the target is a white box with all information provided) or dynamic analysis (where the target is a black box with only minimal information available); however, penetration testing is a process that uses all available means to determine if a system is vulnerable to attack [109]. Fuzzing, or fuzz testing, is another software testing technique in which invalid, unexpected, or random data is provided to the inputs of a program in order to identify problems in software [110]. Code review is the oldest of these methods where human auditors are responsible for identifying defects in software using numerous approaches such as Fagan's Software Inspection, Formal Technical Reviews, Humphrey's Inspection Model, and Structured Walkthroughs [111, 112, 113, 114].

Let the set of all possible analysis methods for identifying potential vulnerabilities be represented by the set X and s_j be a source code artifact from the set of all possible source code artifacts S. We can select a particular analysis method x_i and use it to

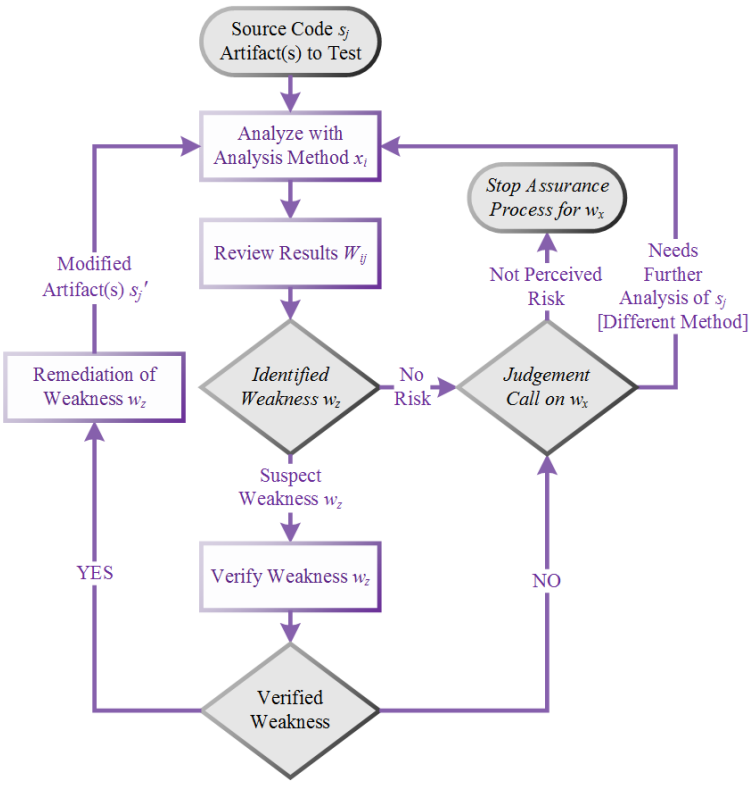

Figure 3.1: Process for verifying ability to exploit.

generate a result set W_{ij} of potential vulnerabilities (i.e. weaknesses) by passing source code s_j to analysis method x_i.

$$W_{ij} = x_i(s_j) \tag{3.1}$$

Thus, each result set W_{ij}, as seen in equation (3.1), consists of zero or more unique weaknesses w_z identified by $x_i(s_j)$.

Once potential vulnerabilities (instances of security weaknesses that may, or may not, be reachable) have been identified by any analysis method, a specific weakness w_z can be inspected with our contract-based runtime monitoring approach in an attempt to verify that the suspected weakness is reachable and exploitable. After a weakness has been verified, it is considered a true positive and remediation can proceed to remove the verified weakness in accordance with its priority. This then takes us into a feedback loop to ensure that the suspected weakness w_z is no longer part of the set of detected weaknesses W_{ij} when analysis method x_i is passed the modified source code artifact $s_j\prime$. However, if a weakness is not verified a judgment call will need to be made to determine if further analysis of w_z is required, by potentially using

other analysis methods from X, before deeming it a false-positive.In some cases, when more than one analysis method reports the same underlying weakness, there may be higher confidence that it is a correct finding (even when not yet verified). Ultimately, each weakness is either placed in a bucket of confirmed true-positives, or discarded false-positives. The only other considered possibility for a reported weakness is to make a judgment that the weakness is not perceived as a risk to security (i.e. a true positive with no security impact relative to a given security policy).

3.3 Prioritizing weakness verification

From the articles of Geer and Schneier, discussed in Section 1.1, we recall that we are interested in not only knowing how many weaknesses are in a particular software artifact but also reducing such a number in a meaningful way. Specifically, reduction can occur by verifying which weaknesses are reachable vulnerabilities and helping developers focus on which defects to address first. When analyzing any software artifact of scale, the size of W_{ij} can be quite large. It is important to have a mechanism that will help prioritize the order in which reported issues are to be evaluated. From the earlier analogy of Geer, this would be akin to addressing one species at a time (starting with the most dangerous predators in the pond).

Vulnerabilities have associated risk related to how much impact there is to the system if exploited, and the likelihood that the vulnerability will be exploited. MITRE has produced the Common Vulnerability Scoring System (CVSS) as one approach to prioritize risk for exploitable vulnerabilities.

Weaknesses are not verified vulnerabilities, rather, they are assertions that a defect in the code may lead to a vulnerability. MITRE has also provided the Common Weakness Scoring System (CWSS), however, most assessment tools use their own scoring systems (such as the Fortify Priority Order (FPO) used by Fortify SCA). The priority used by Fortify is also similar to the severity and likelihood used by MITRE's CAPEC attack patterns. The five weakness categories above, for example, have a default FPO value of *critical* based upon a risk calculation involving impact of exploitation and likelihood of exploit (critical, high, medium, low) [115].

Addressing weaknesses from most critical to the least, based upon impact and likelihood, the risk to the system is reduced incrementally until all identified issues have been resolved and $W_{ij} = \varnothing$. When $W_{ij} = \varnothing$ is reached, we still cannot state that a system is secure and is no longer vulnerable. We can merely state that the

system is now secure against all exploits that could have been conducted leveraging the vulnerabilities, based upon the underlying weaknesses, identified using the current capabilities of the analysis methods employed from the set X. It is entirely possible that a new type of weakness could be discovered in the future which no existing technique or method was capable of detecting at an earlier time.

3.4 Integrating Security Monitoring in a SDLC

Our proposed method for CB_SAMF can be used against existing products; however, it would be more efficiently applied as part of an overall SDLC strategy. Since source code is continuously evolving during development and vulnerability detection mechanisms also improve, it is important that the detection, verification, and mitigation of underlying weaknesses is conducted iteratively during systems development life cycles. In this section we introduce some of the activities that can be integrated into existing SDLCs, explore some of the available modeling practices for security, and propose our approach for integrating CB_SAMF into a more security focused SDLC.

Vulnerabilities in particular span multiple conceptual domains and as such our primary focus is on applying contracts during implementation rather than production; however, many design and operational vulnerabilities can also be targeted by contracts during the testing phase of a development life cycle.

3.4.1 Secure Software Development Life Cycle

Security activities, such as those depicted in Figure 1.3, can be integrated into the iterative and recursive SDLC. These security activities provide various advantages and disadvantages. During specification and design we can identify many potential security threats relating to the security requirements through the use of misuse cases and attack trees. These potential threats can then be used during design inspection to help prevent vulnerabilities from forming. During implementation, static analysis can be used to augment code inspection for potential vulnerabilities; however, static analysis is known to have potential high rates of false positives [87]. The output from the security analysis and design activities can also be used in the derivation of security testing procedures and test cases which can be used for penetration testing during testing and deployment. We also know from experience that security attacks can still occur once a product has been shipped and placed in production. It is therefore

warranted to also have tools such as firewalls, antivirus, intrusion detection systems (IDS), intrusion prevention systems (IPS), web application firewalls (WAF) and others deployed in the production environment.

Mechanisms for detecting and preventing attacks in a live environment, or the maintenance phase of the SDLC, are currently necessary to limit the potential damage to consumer systems. These technologies, such as web application firewalls and intrusion detection/preventions systems, will never completely eliminate the security risk since they do not remove the underlying cause of the vulnerabilities from the systems. If any of these mechanisms should fail at runtime, and the underlying defect has not been fixed, then a successful exploit and attack can still occur. CB_SAMF is primarily intended to be employed during design through testing phases of the SDLC for the detection of events which contradict the security policy for the system. Artifacts created and analyzed during design provide targets and security knowledge for the creation of contracts during implementation and testing phases. The security policy, risk assessment, and requirements documents (which may involve standards compliance) all contribute to the identification of potential security assertion targets for a given system. Through the use of security assertions, in the form of contracts, CB_SAMF can verify the existence of a vulnerability for immediate removal. Central to our discussion of contracts and runtime are the following three terms:

Breakpoint: An intentional interruption, or pausing, location in a program during execution typically used for debugging purposes.

Assertion: A predicate that is always expected to be true at a particular location in a program.

Contract: An agreement between two or more parties that states one or more contractual obligations between the parties.

Contracts within our CB_SAMF specify assertions that must remain true at a particular location which essentially corresponds to a software breakpoint location.

3.4.2 Modeling

Using popular specification modeling notation, such as UML, many different diagram types can be used to visualize and conceptualize the functionality of software. Misuse cases, in particular, are created early during the SDLC in order to identify potential

security threats to a system and are then revised throughout the SDLC. Misuse cases are further refined into attack trees, or attack patterns, by identifying the security violation scenarios exposed. Each path from the root to a leaf node in an attack tree will identify possible violation scenarios [116]. Both misuse cases and attack trees are used to further our understanding of potential security threats against a system. Each threat will have an associated probability or likelihood of occurring. In order for a threat to be realized as an attack the associated vulnerabilities must exist in the system. Many vulnerabilities can be represented as rules, and as such, a certain subset of vulnerabilities can be identified in a system through their sequence diagrams.

Exploitation of vulnerabilities contradict the specified acceptable use of a system. Acceptable use of a system, network, or application is usually specified in the form of a security policy specification document. Risk assessment and requirements gathering also contribute to the identification of security assertions. Ultimately, it is the requirements document that specifies the security requirements of the system. These policies and requirements are typically written in natural language; therefore, rules or assertions need to be extracted from these documents before they can be inserted into an assertion framework. It is then the responsibility of the framework to monitor these assertions in real-time and take appropriate measures when a breach is detected.

3.5 Specific Weaknesses

Many of the most commonly occurring critical vulnerabilities observed in industry are related to improper, or lacking, input validation. In general, weaknesses that can be grouped under the general classification of *input validation* are prime candidates for verification using standard inputs since user-controlled values provided as input at runtime ultimately lead to successful attacks if they contain the necessary outliers to exploit an underlying weakness. For each of the following input validation weakness types we will consider the weakness definition, associated CWE, the attack pattern, possible detection methods, and implementation mitigations. For attack patterns, MITRE provides the Common Attack Pattern Enumeration and Classification (CAPEC) as a publicly available catalog and schema for describing related attacks[3]. Each attack pattern, as of CAPEC v2.8, provides a description of the common elements and techniques used for a given attack.

[3]https://capec.mitre.org/index.html

3.5.1 Format String Vulnerability

Description: All or part of the input received from an externally influenced component is used as the format string in printf-style functions. Use of user input as a format string can lead to buffer overflows, reading/writing of the user-mode stack, and data representation issues.

Weakness: CWE-134: Use of Externally-Controlled Format String.
Attack Pattern: CAPEC-135: Format String Injection
Severity: CAPEC-135: High
Likelihood: CAPEC-135: High

Detection: Static analysis, dynamic analysis, manual analysis.
Mitigation: Input validation to ensure only static strings are used for format strings and that the proper number and type of arguments are used.

3.5.2 Resource Injection/Path Manipulation

Description: All or part of the input received from an externally influenced component is not properly neutralized before being used as an identifier for a resource that may fall outside the intended sphere of control. Improper neutralization may lead to the access or modification of protected resouces.

Weakness: CWE-99: Improper Control of Resource Identifiers ('Resource Injection')/CWE-73: External Control of File Name or Path ('Path Manipulation')
Attack Pattern: CAPEC-240: Resource Injection/CAPEC-76: Manipulating Input to File System Calls
Severity: CAPEC-76: Very High
Likelihood: CAPEC-76: High

Detection: Static analysis, dynamic analysis, manual analysis.
Mitigation: Input validation involving white-listing of acceptable resources. Avoid black-listing.

3.5.3 OS Command Injection

Description: All or part of an OS command is constructed from externally influenced input without properly neutralizing special elements that could modify intended behavior. Improper neutralization of the input could lead to the execution of unexpected and potentially dangerous commands.

Weakness: CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection').
Attack Pattern: CAPEC-88: OS Command Injection
Severity: CAPEC-88: High
Likelihood: CAPEC-88: High

Detection: Static analysis, dynamic analysis, manual analysis.
Mitigation: Input validation, output encoding to escape any special characters, white-listing of valid characters, and properly quote any arguments.

3.5.4 SQL Injection (SQLi)

Description: All or part of the input received from an externally influenced component is not properly neutralized for special elements that could modify the intended SQL command. Improper neutralization can lead to bypassing security controls, insertion of additional SQL statements which modify the database, and possibly even execution of system commands.

Weakness: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Attack Pattern: CAPEC-66: SQL Injection
Severity: CAPEC-66: Very High
Likelihood: CAPEC-66: High

Detection: Static analysis, dynamic analysis, manual analysis.
Mitigation: Input validation, output encoding to escape any special characters, white-listing of valid characters, and properly quote any arguments.

3.5.5 Basic Cross-Site Scripting (XSS)

Description: All or part of the input received from an externally influenced component is not properly neutralized for special characters { <, >, &, ", ', / } that could be interpreted as web-scripting elements when passed to components that process web pages. Improper neutralization of special characters can lead to the injection of malicious scripts into trusted web sites.

Weakness: CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS).
Attack Pattern: CAPEC-63: Simple Script Injection
Severity: CAPEC-63: Very High
Likelihood: CAPEC-63: High

Detection: Static analysis, dynamic analysis, manual analysis.
Mitigation: Input validation of all parts of the request which includes white-listing of valid input characters as well as output encoding. Avoid black-listing.

3.6 Summary

Separating the inherent phases of vulnerability creation into separate components requires specific terminology to help avoid confusion. In this chapter we defined the difference between issues, defects/weaknesses, vulnerabilities, exploits, and attacks before using these terms to explore five specific critical weakness types used in this dissertation. Furthermore, we introduced a general approach for identifying weaknesses and verifying whether or not they are a root cause for a vulnerability which may be exploited during an attack. Finally, we discuss how dealing with the complexity and volume of detected weaknesses can be handled through prioritizing of weaknesses from most critical to least using metrics such as Fortify's FPO or MITRE's CWSS before reviewing how CB_SAMF can be integrated into an augmented SDLC with the assistance of modeling approaches such as attack trees.

Chapter 4

CB_SAMF

The world is not dangerous because of those who do harm but because of those who look at it without doing anything.
-Albert Einstein

Now that we have introduced the core terminology and mechanisms for identifying weaknesses and verifying vulnerabilities, and presented how CB_SAMF can be integrated into an existing SDLC, we will now switch our focus to introducing our proposed contract model. We will begin by presenting our proposed model through the syntax, semantics, and a case study regarding how Contract-Based Security Assertion Monitoring (CB_SAMF) can be applied to the lowest levels of a software stack in order to verify the exploitability of two different weaknesses. Finally, we discuss how our model is realized by prototyping our solution within a Linux environment.

4.1 Model for Security Assertion Monitoring

The use of contracts in software engineering is not a new idea [57, 58, 59, 60]. When used for security, however, we must look outside of the basic pre and post conditions that are often used when implementing systems using contracts and look carefully at what properties need to be specified in a contract to improve security. Many pre and post conditions deal more with robustness than security. Historically, the pre condition specifies when it is appropriate to call a particular feature (function/method), while a post condition specifies what is true after a particular feature is called (what has

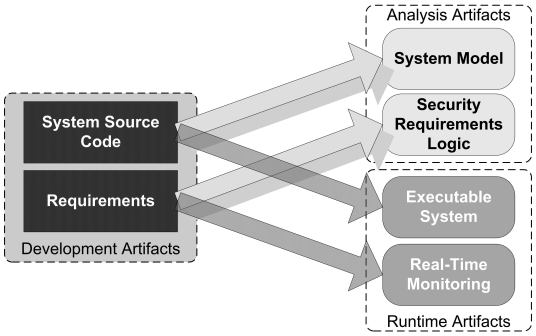

Figure 4.1: Analysis, development and runtime artifact relationships. From the source code and requirement artifacts we extrapolate both analysis artifacts and runtime artifacts.

been accomplished by the function/method). This refers to the concept that the system must be in a specified state before the call and that the feature will leave it in another specified state. We desire a form of contract, derived from security requirement assertions, that can be used for monitoring in addition to being used for specification and design. The contract should identify environmental and system security conditions necessary for the generation of probes; these probes can then monitor the security assertions for violations during runtime as in Figure 4.1. If, or when, a violation occurs we know that one or more vulnerabilities have been found and we can then deploy countermeasures.

Our definition of contract, called Contract-Based Security Assurance Monitoring (CB SAMF), binds the caller and callee to deal with additional properties involving timing, property values, and other events. For example, consider a contract that is specified for a callee X that provides something to be consumed by a caller Y. The contract guarantees that X has fulfilled the post condition(s), provided that Y has satisfied the pre condition(s). Thus, the contract provides protection for both parties. The caller is protected from the callee since the post conditions have been guaranteed by the callee. The callee is protected from the caller since the pre conditions have been guaranteed by the caller.

Contracts, as proposed by Meyer, are not suitable for security monitoring. For example, under normal contracts, a false pre condition does not guarantee that the system will not process the input. It may still allow certain types of attacks such as a buffer overflow. The require, guarantee, and references fields of the contract, that correspond to pre, post, and invariants, do not handle all of the necessary attributes of security defects. We propose the addition of several new contractual fields including

context, *history*, and *response*. Context is required since the basic reliability contract above does not factor environmental influence. History is required for state transitions since security vulnerabilities are often complex and are sometimes the result of a series of actions that may occur in parallel. Both context and history can be useful when dealing with DoS and race-condition vulnerabilities. Finally, response is required so that we can choose how a particular assertion is handled when an exploitation of a vulnerability is detected.

Our form of contract includes the following fields:

Requirements: In the form of pre conditions (PRE)
Guarantees: In the form of post conditions (POST)
References: In the form of invariants (INV)
Context: In the form of relevant environmental information (CONT)
History: In the form of some knowledge keeping construct (HIST)
Response: In the form of a reactive measure (RESP).

4.1.1 Syntax

Work done by Barringer *et al* on program monitoring and rule-based runtime verification has exposed interesting results [43, 44]. Specifically, the work on linear temporal logic (LTL) and program states has been core to several attempts towards runtime verification and is a promising candidate for the notation of our contracts. While we could implement our rules informally, as discussed earlier, we reuse the syntax of EAGLE for its expressive power of expressions, functions, rules, monitors, and a subset of LTL in order to state that certain events either have, or have not, occurred in the past or in the future. We extend the base syntax with the ability to take reactive measures (responses), keep track of environmental data and historical events in order to track the context of a potential vulnerability (and the data relevant to them), and associate our contracts with specific breakpoint locations in the code to enable inline runtime verification of security vulnerabilities without the need for recompiling the target application. The *min* and *max* quantifiers allow us to reason as to whether a contract was violated when a probe is removed. Our approach is only semi-formal in the sense that we do not monitor every operation between two events that we are interested in since that can be derived statically (offline) to lessen the performance

impact at runtime. A formal verification technique, such as theorem proving or model checking, would need to check all states.

We do not want to specify the properties of an entire program. Rather, we aim to specify security properties that must hold at a specific state transition in a program. These security properties take the form of contracts specifying assertions at a specific location (breakpoint).

Each contract (C) will contain a breakpoint (B) and one or more assertions (A). A breakpoint identifies a monitoring location, or symbol, in the target application. For example, a contract should be able to specify a target function in a program which affects the state of an assertion. The assertion is a rule that must remain true at the breakpoint. Each assertion has associated with it zero or more of the security contract extensions (E) mentioned above (context, history, and response).

An assertion can take on one of the following three forms: pre condition (PRE), post condition ($POST$), or invariant (INV). We do not represent the assertions types separately since they all take the same form. Each assertion is composed of zero or more rules (R), relating to the target (remember the breakpoint B), and zero or more monitors (M). The rules, monitors, and extensions are individually named (N) while parameters are typed (T) and can be either a primitive type or a boolean valued formula (F). A rule specifies a property of the state of the program that needs to remain true, while a monitor enforces one or more rules. The quantifiers *min* and *max* represent liveness and safety properties respectively and are important for the boundary cases of a monitor trace [19, 44]. If specifying a contract with a liveness property then we are stating something good will eventually happen, however, if the condition is not met before monitoring terminates then the generated probe will return a negative (indicating "good thing" has not yet been observed). Specifying a safety property is stating that something bad never happens and if the condition is not met before monitoring terminates then the generated probe will return a positive (indicating "bad thing" was never observed during monitoring period).

The body of every rule and monitor is specified as a boolean valued formula of the syntactic category *Form*. This notation is derived from linear temporal logic (LTL) and can be considered an extension of the EAGLE syntax for a specification that was proposed by Barringer *et al* [43, 44]. Specifically, the rules for A, R, M, T, and F are derived directly from the EAGLE model.

We have also defined possible extended behaviours for context, history and response elements and may extend these in the future. These contractual clauses, which consist

of special named functions, are not part of the EAGLE framework proposed by Barringer *et al.* [44, 43]. Context may specify environmental or resource information that is needed by the contract. History may contain trace data (history keeping data structure), or statistically relevant information (history related algorithm), for the contract and can maintain state information from tracing activities or statistical computations such as average use of a resource. Finally, response may specify an action to perform when an assertion is violated. Possible responses (*resp*) include the following: core=produce a core dump, term=terminate the task, kill=kill the task, log=produce an audit report for the event, and reboot=reboot the system to a stable state. Each contract may be instantiated using the following grammar:

$$
\begin{aligned}
&C := B\,(A\{E\})\,\{A\{E\}\};\\
&\quad E := \{CONT\} \mid \{HIST\} \mid \{RESP\};\\
&\quad A := \{R\}\{M\};\\
&\quad R := \{\underline{\max}|\underline{\min}\}\ N(T_1x_1, ..., T_nx_n) = F;\\
&\quad M := \underline{\text{mon}}\ N = F;\\
&\quad T := \underline{\text{Form}} \mid \textit{primitive type};\\
&\quad B := \textit{symbol} \mid \textit{HEX address};\\
&\quad F := exp|\underline{\text{true}}|\underline{\text{false}}|\neg F|F_1 \wedge F_2|F_1 \vee F_2|F_1 \rightarrow F_2|\odot F|\bigcirc F|\\
&\qquad\quad F_1 \cdot F_2|N(F_1, \ldots, F_n)|x_i;\\
&\quad CONT := \underline{env}\ N \mid \underline{res}\ N;\\
&\quad HIST := \underline{dstruct}\ N \mid \underline{alg}\ N;\\
&\quad RESP := \underline{resp}\ N;
\end{aligned}
$$

From this definition of contracts it is possible to use multiple separate monitors or redirect multiple rules to the same monitor.

4.1.2 Semantics

The following is a short comparison of different notations for semantic definition. We then provide a semantic definition for the key elements of our model.

Denotational semantics, specifying what the program means, provides a denotation (a literal meaning) in a mathematical domain for each string in a language via the definition of semantic functions.
Axiomatic semantics, specifying what properties the program has, provides meaning through the definition of a set of laws or axioms ($x = y$) that the language must satisfy.
Operational semantics, specifying what the program does, provides a set of rules (x $\rightarrow$ y) that provide a possible evaluation or execution of a program written in the language.

We provide an axiomatic meaning that will be defined using a satisfaction relation $\models$, between runtime execution and specifications of the form:

$$\sigma \models C$$

that describes the situation where state σ satisfies a given contract C where a minimal contract consists of $C := RM$. Therefore, given σ and a contract C, we define:

$$\sigma \models \text{RM iff } \forall(\underline{\text{mon}}N = F) \in M.\sigma \models_R F$$

Where the state during runtime satisfies the specified assertion, consisting of zero or more rules (R) and monitors (M), if and only if, for all monitored formulas there exists a monitored state that satisfies R. Furthermore, the purpose of the contract and an associated breakpoint are defined respectively as follows:

Contract:
Form: contract (C)
Purpose: To be able to specify an agreement between two or more parties (caller and callee) that states one or more contractual obligations between the parties. The contract takes the form of assertions which when violated will invoke one or more monitors.

Breakpoint:
Form: symbol or virtual address (B)
Purpose: To be able to specify an intentional interruption, or pausing, location in a program during execution. Breakpoints are an address or symbol, to which a rule or monitor is associated, that when hit during execution will evaluate the associated rule or monitor.

For satisfaction, we represent a relation between state traces (inline) and contracts. A trace satisfies a contract if the trace satisfies each monitored formula for all given states. For example, all input validation problems are tied to two primary states. First, when the tainted data entering the system contains malicious content that could lead to an exploit (the source). Second, the malicious content that entered at the source remains dangerous and is consumed by the function which is vulnerable to such input (the sink). If these two properties hold at the sink, then the exploit is possible. If the first property holds and the second does not, then the exploit does not occur.

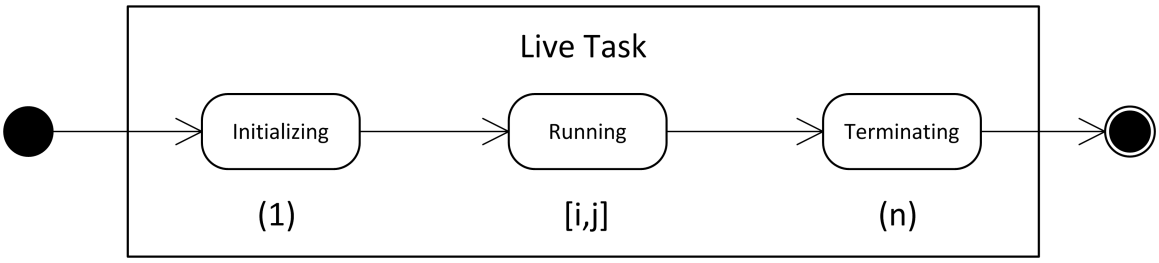

Figure 4.2: Possible states that a monitored program can be in during runtime. State 1 is the initial state, n is the final state, and $[i, j]$ are the intermediate states.

There are several possible state traces from runtime monitoring for a given contract as shown in Figure 4.2. We consider $\sigma^{i,j}$ to denote a sub-trace of states from state i to state j, state 1 to be the first state, and state n to be the final state (terminating):

$$\sigma^{[1,i]}, beginning\ state\ to\ later\ intermediate\ state\ (non-terminating) \tag{4.1}$$

$$\sigma^{[i,j]}, intermediate\ state\ to\ later\ intermediate\ state\ (non-terminating) \tag{4.2}$$

$$\sigma^{[j,n]}, intermediate\ state\ to\ final\ state\ (terminating) \tag{4.3}$$

$$\sigma^{[1,n]}, beginning\ state\ to\ final\ state\ (terminating) \tag{4.4}$$

If a contract is violated, in that it has not been satisfied at a particular state and the rule returns false, we can certainly claim that the assertion has not been held in at least one of the cases above. If a contract is not violated, however, then we need to consider each of the four permutations above to determine if an assertion held.

The notion of termination is important for rules which have been stated to be minimal or maximal. If a monitored task terminates (beyond terminating state n), or monitoring stops, and the corresponding rule is stated to be maximal the associated rule evaluates to true (safety property/nothing bad ever happens); alternatively, if the rule

is stated to be minimal the associated rule evaluates to false (liveness property/something good eventually happens). However, considering most weakness categories will have to do with maximal conditions, we must also state that it is entirely possible that insufficient data coverage was used as input during testing and we should ultimately consider such a case as *undetermined*.

Formula:
Let us set the stage by providing the typical meaning of a boolean *formula* from propositional and temporal logic for our contract which comes from the work of Barringer *et al* [43]. While our method evaluates assertions based on the present state and history information, for the purpose of reasoning about temporal properties, we use the notation $\sigma^{[i,j]}$ to refer to the series of states from i to j where the first state is 1, the terminating state is n, and the length of the trace is $|\sigma| = n$. Keeping in mind that the states we care about are roughly equivalent to the events triggered by our probes, as such, one may also think of these as finite traces. Semantics for the boolean valued formula *(F)*, specified as satisfaction relations, are as follows:

$$
\begin{aligned}
\sigma \models_R exp,\ & iff\ [exp]_\sigma == true \\
\sigma \models_R \underline{\text{true}},\ & always \\
\sigma \not\models_R \underline{\text{false}},\ & never \\
\sigma \models_R \neg F,\ & iff\ \sigma \not\models_R F \\
\sigma \models_R F_1 \wedge F_2,\ & iff\ \sigma \models_R F_1 \wedge \sigma \models_R F_2 \\
\sigma \models_R F_1 \vee F_2,\ & iff\ \sigma \models_R F_1 \vee \sigma \models_R F_2 \\
\sigma \models_R F_1 \rightarrow F_2,\ & iff\ whenever\ \sigma \models_R F_1\ then\ \sigma \models_R F_2 \\
\sigma \models_R \odot F,\ & iff\ \sigma' \models_R F,\ where\ \sigma' = \sigma_{n-1}\ if\ n > 1,\ otherwise\ \sigma' = \sigma \\
\sigma \models_R \bigcirc F,\ & iff\ \sigma' \models_R F,\ where\ \sigma' = \sigma_{n+1}\ if\ n < termination\ state, \\
& \quad otherwise\ \sigma' = \sigma \\
\sigma \models_R F_1 \cdot F_2,\ & iff\ \exists\ j\ s.t.\ i < j < |\sigma|\ and\ \sigma^{[1,j-1]} \models_R F_1\ and\ \sigma^{[j,|\sigma|]} \models_R F_2 \\
\sigma \models_R N(F_1, \ldots, F_n),\ & iff\ 1 < i < |\sigma|\ then:\ \sigma \models_R F[x_1 \mapsto F_1, \ldots, x_n \mapsto F_n] \\
& \quad where\ (N(T_1 x_1, \ldots, T_n x_n) = F)\ \in\ R)
\end{aligned}
$$

For the above semantics, derived from EAGLE [43] where EAGLE evaluates across all states, our inline approach for CB_SAMF evaluates on the current state. As such, an *exp* (atomic formula) is evaluated in the current state. Boolean constants and connectives have their usual meanings. The next-state and previous state formulas will evaluate to true if the formula F continues to hold in future and historical states (fully evaluated when monitoring ceases for the contract). Concatenation is supported if a trace can be divided into two sub-traces where F_1 holds on the first sub-trace and F_2 holds on the second sub-trace.

With the formula semantics used by the rules and monitors discussed in terms of satisfaction relations above, we will switch to Hoare [117] rules of inference to present the straightforward meaning of contractual top level assertions. Rules of inference are interpreted as the item below the line as being inferred from item above the line. First, a contract has one or more assertions associated with a breakpoint.

Assertion:
Form: requirements, guarantees, references
Purpose: To be able to specify a predicate that is always expected to be true at a particular breakpoint/location in a program (B). Assertions take the form of rules (R) and monitors (M) .
Types: PRE, POST, INV.

Each assertion can be one of three possible forms {*Requirements*, *Guarantees*, *References*} as follows:

Requirements:
Form: preconditions (PRE)
Purpose: To be able to state that something must be true (R) prior to an operation (B), which if violated can indicate an attack. A pre-condition assertion is a boolean expression that must hold true (M) at the beginning of a state transition and is typically an assertion that must be satisfied by the caller/consumer.

$$\frac{}{\{P\}S_1\{true\}} \tag{4.5}$$

Guarantees:
Form: post conditions (POST)
Purpose: To be able to state that something must be true (R) after an operation (B), which if violated can indicate an attack. A post-condition assertion is a boolean expression that must hold true (M) at the end of a state transition and is typically an assertion that must be satisfied by the callee/provider.

$$\frac{}{\{true\}S_1\{R\}} \tag{4.6}$$

When a contract is written it will often be necessary to combine both requirements and guarantees for statement(s) Q. The proof is straightforward using the composition rule:

$$\frac{\{P\}Q\{true\}, \{true\}Q\{R\}}{\{P\}Q\{R\}} \tag{4.7}$$

References:
Form: invariants (INV)
Purpose: To be able to state that something that is true (R) prior to an operation remains true during and after the operation completes (B), which if violated can indicate an attack. In other words, an invariant assertion is a boolean expression that must always hold true (both pre and post) and typically occur as loop invariants (M). The proof is that of iteration from Hoare Logic:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}while\ B\ do\ S\{P \wedge \neg B\}} \tag{4.8}$$

For Hoare triples, it is also worth discussing the notion of partial and total correctness. Partial correctness of a Hoare triple $\{P\}S\{Q\}$ is said to be partially correct "if S executes in a state in which P holds, then it terminates in a state in which Q holds (unless it aborts or runs forever)". Total correctness of a Hoare triple occurs "if S executes in a state in which P holds, then it terminates in a state in which Q holds".

Finally, each assertion can be combined with zero or more extensions (E), represented as specialized named functions (covered under the satisfaction relations above), for maintaining context or history, and for registering a response to be invoked if an assertion is violated.

Context:
Form: relevant environmental information (CONT)
Purpose: To be able to relate the state(s) of the application to environmental properties which when combined can indicate an attack (exploitation of vulnerability). Context assertions are universal assertions that must hold true as PRE, POST, or INV.

History:
Form: knowledge keeping construct (HIST)
Purpose: To be able to relate two or more events which independently do not cause concern but when combined can indicate an attack (exploitation of vulnerability). Historical properties are tracked for past-time or future time temporal assertions that must hold true as PRE, POST, or INV.

Response:
Form: reactive measure (RESP)
Purpose: To be able to respond appropriately to a realized attack (exploitation of vulnerability). Responses are reactive rules, represented as specialized named functions that always return true, which cause a monitor to react to the failure of an assertion in PRE, POST, INV, CONT, or HIST.

With our inline monitoring using probes, in contrast to EAGLE where state is maintained by an observer, the algorithm used to evaluate whether or not a contract is violated consists of the logic of the contract itself with the state being maintained by the probes through context and history. For any assertion which evaluates to false, during execution, any registered reactive measures will execute. When monitoring is stopped, by removing the probes, the resulting status of the contract(s) will be reported as being violated or not violated (with liveness and safety properties taken into account).

4.2 Case Study

Attacks against systems are ultimately dependent on the existence of vulnerabilities, and the underlying weaknesses, which were inserted either intentionally or through human error. To show how our model can be implemented we will next demonstrate that both an exploitable logic error (resulting in a buffer overflow) and a concurrency

error (resulting in a denial of service) can be identified and monitored. In both examples we will assume that the normal phases of SDLC have been adhered to and that our additional steps will expose the vulnerabilities we wish to verify. Focus will be given only to those steps in the SDLC that are relevant to our security contracts. These initial experiments were conducted within the following environment on a Dell Vostro 200:

- RAM: 2GB
- Hard drive: 500GB Seagate Barracuda 500GB 3.5" Desktop Drive SATA
- Processor: Intel Core2 Duo 1.6GHz
- Operating System: Fedora 15
- Kernel version: 2.6.40.4-5.fc15.i686

The remainder of this section consists of a discussion of vulnerability identification and the need for verification, a summary of system requirements for the case study, an overview of the security requirements analysis and design, the creation of CB_SAMF contracts for the identified vulnerabilities by first showing LTL expressions for the vulnerability assertions, and concluding with how the contracts verify the vulnerabilities at runtime.

4.2.1 The Need for Verification

We have experimented with various methods for identifying vulnerabilities including both code review and automated static analysis. While code review requires a great deal of human time, knowledge and effort, static analysis requires only resources and time. The primary disadvantages of code review are the cost of human time and the variations in reviewer experience. The primary disadvantage of static analysis is the required auditing time to review issues and eliminate false positives which can occur at a relatively high rate. Static analysis is only capable of finding vulnerabilities that can be specified by rules, whereas human led code review can identify additional defects derived from past experiences. Alternatively, static analysis can greatly reduce auditing time by indicating areas of code that likely contain defects.

For example, one of the more capable static analysis tools for security, which we discussed earlier, is the Static Code Analysis (SCA) suite by Fortify Software which is now part of Hewlett Packard Enterprise [87]. In a realistic example of a sizable project related to telecommunication software engineering, the Linux kernel (version 2.6.31, patched with utrace) was put through static analysis using a Lenovo(R) Core(TM)2

Duo, 2GHz machine, using 2GB of RAM, running Fedora Linux 10, and SCA 5.2. In this early experiment, we chose the Linux kernel since it provides the basis of Carrier Grade Linux (CGL) compliant distributions.

While the 2.6.31 Linux kernel consists of more than 12 million lines of code (LOC), only a subset is used in the creation of an actual kernel based on what features have been configured for the kernel prior to build.[1] Our configuration included 2,609 of a possible 3,422 configuration settings and resulted in a binary kernel image (vmlinux) with a size of 111MB.

Static analysis using Fortify consumed a considerable amount of CPU time and RAM, resulting in 3.9GB of storage being consumed for intermediate files (only considering SCA generated files), which were then used to generate a 37MB report file. The report file was then analyzed using the Audit Workbench provided by Fortify Software. In total the static analysis was able to identify 8,781 issues, in 9,366 files, consisting of 1,422,303 LOC. A summary of the findings is displayed in Table 4.1.

Issue Category	**Hot**	**Warning**	**Info**	**Subtotal**
Buffer Overflow	53			53
Buffer Overflow: Format String		284	1	285
Buffer Overflow: Off-by-One	29			29
Dangerous Function: Strcpy			1,434	1,434
Dead Code			276	276
Format String			130	130
Format String: Argument Type Mismatch		6		6
Insecure Compiler Optimization: Pointer Arithmetic			1	1
Insecure Randomness			4	4
Memory Leak		1		1
Missing Check against Null		11		11
Null Dereference		1,169		1,169
Out-of-Bounds Read		100		100
Out-of-Bounds Read: Off-by-One		46		46
Out-of-Bounds Read: Signed Comparison		2		2
Password Management: Hardcoded Password	7			7
Password Management: Password in Comment			63	63
Poor Style: Redundant Initialization			87	87
Poor Style: Value Never Read			2,235	2,235
Poor Style: Variable Never Used			285	285
Redundant Null Check		1,107		1,107
System Information Leak		11		11
Type Mismatch: Negative to Unsigned			154	154
Type Mismatch: Signed to Unsigned			742	742
Unchecked Return Value			51	51
Uninitialized Variable			488	488
Weak Encryption	4			4
Totals	93	2,737	5,951	**8,781**

Table 4.1: Summary of issues (potential weaknesses) identified by Fortify SCA 5.2 of a Linux 2.6.31 compiled kernel. The three middle columns indicate the severity of the issue from high to low.

[1] http://www.h-online.com/open/features/The-Next-Round-The-new-features-of-Linux-2-6-31-746619.html

While there are relatively few "Hot" issues, all of the reported issues still require validation to ensure that they are in fact vulnerabilities. Of particular interest are the */net* (1,925 issues) and */drivers/net* (483 issues) directories in the kernel source tree which represent networking protocols and networking device drivers respectively. The potential issues identified within these two directories indicate vulnerabilities that are directly related to telecommunications. The */net* directory contains suspected vulnerabilities ranging from "Hot" to "Info", whereas */drivers/net* ranges from "Warning" to "Info".

We expect a relatively high false-positive rate for the potential vulnerabilities; however, we need to verify the results from both static analysis and code review processes. In the following subsections we will show how two different vulnerabilities, one that could be detected by the above static analysis tool (focused on a "Hot" vulnerability) and one that would not be detected, can be verified using our CB_SAMF.

4.2.2 Summary of System Requirements

First, assume that a new system has been requested. It requires an administrative tool be developed along with a device driver for a new piece of telecommunication hardware which will run under the Linux operating system. The specification requires the tool for ease of use by administrators, while the driver provides the underlying functional support for the hardware (system call hooks and interrupt handling).

To provide the underlying functionality for the administrative tool we must pass information from user-space to kernel-space. While it is possible to control settings using the `ioctl` function, it is often the case that the state of a driver is modified through either kernel or module parameters. These settings may also be manipulated through virtual file systems such as *sysfs* and *procfs*. The device driver will thus have one or more file descriptors made available through procfs virtual files to exchange information between user space and kernel space to support the administrative tool. Each virtual file, for example /proc/target, will require two hooks for reading (`read_target`) and for writing (`write_target`). Once the driver is loaded and initialized, any read operations will invoke `read_target` and any write operation will result in a call to `write_target`.

Most device drivers will use resources that must be protected from concurrent access. Since most modern systems now have multiple core, or symmetric multiprocessor (SMP), architectures it is likely that spinlocks will be used to protect any

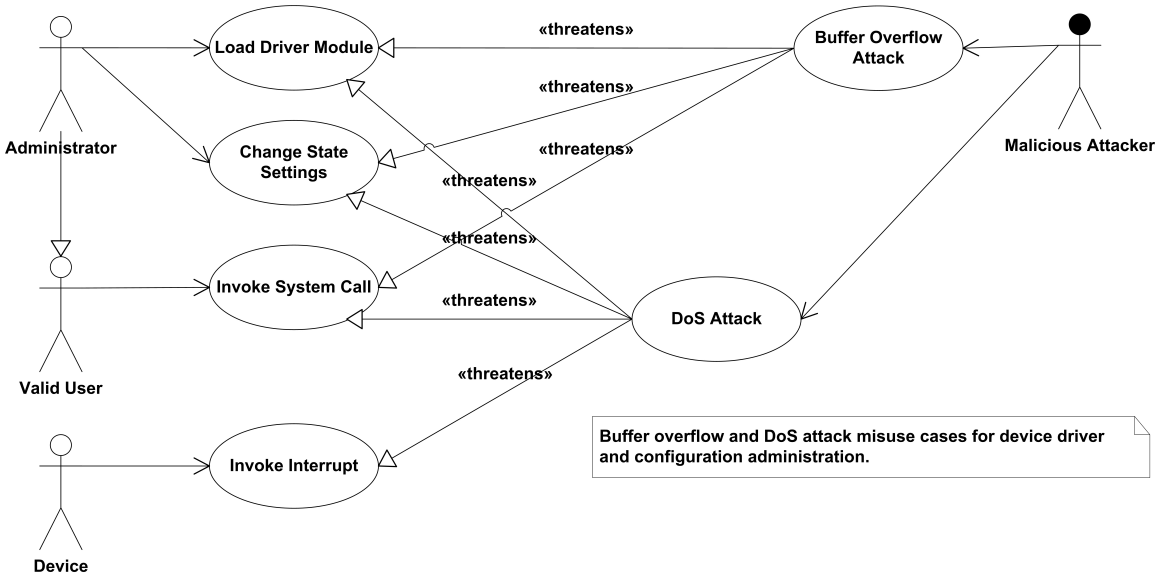

Figure 4.3: Use cases for device driver and device configuration along with misuse cases for buffer overflow and DoS attacks.

shared resources in the driver. Spinlocks are similar to traditional semaphores; however, when contention arises during execution a task which blocks for the shared resource will busy wait rather than be placed in a wait queue.

The above excerpt of requirements will be used to identify potential threats.

4.2.3 Security Requirements Analysis and Design

The security policy document specifies many general rules regarding what it means to be secure for an organization, system, or other entity. During our analysis phase we would extract from the security policy (and from other documents and stake holders) the necessary information to put together the security requirements for this system. The security requirements are necessary in order to evaluate if the system is secure. From the requirements documents we can determine the valid and invalid actors in a system and the necessary use cases. During use case analysis we can also generate our misuse cases such as the one highlighting potential buffer overflow and DoS attacks against our device driver in Figure 4.3. We would have other diagrams depicting other threats and software artifacts such as the administrative tool for the device.

The monolithic kernel is the most sensitive environment in the Linux operating system. Modules are typically used to add device drivers and other functionality to the monolithic kernel dynamically at runtime. Modules can only be installed by a user with root privileges. Once installed, the functionality of modules can be utilized

by multiple users until the module is unloaded. The module examples we give below show how buffer overflows and DoSs occurring in the kernel are severe security risks.

Once we have the collection of misuse cases for the system we can then further our understanding of the threats by creating attack trees or patterns. Attack trees are conceptual diagrams that provide a methodical way to expose threats to a system and the attack steps required to realize those threats. These attack trees can either be realized as a tree graph or as a list. Attack trees are read from the leaf-nodes up to the root, where the root of the tree is a compromised system.

While an attack tree for an entire system tends to be very large, two sub trees of software vulnerabilities showing the threat of a buffer overflow and a DoS to our system are depicted below. These two sub trees depict how an attacker, or tester, could go about locating and exploiting the named vulnerabilities.

1. Exploit a software vulnerability.
   (a) Locate a buffer overflow weakness.
      i. Apply static analysis to source code to locate potential buffer overflows.
      ii. Apply manual code inspection to locate potential buffer overflows.
      iii. Apply fuzz testing on application to locate potential buffer overflows.
   (b) Locate a race condition weakness.
      i. Apply static analysis to source code to locate potential race conditions.
      ii. Apply manual code inspection to locate potential race conditions.
      iii. Study design documents to locate potential race conditions.

Once the security related tasks have been completed during the analysis phase the created artifacts are then used during design to make appropriate design decisions. Risk analysis, with a focus on security, is able to prioritize the threats identified by the (mis)use case diagrams and attack trees. During design the architects have selected to implement a device driver in C (the only programming language for device drivers in Linux) and created a command line interface for testing before creating the administrative tool. At this point the selection of language has some implications for security. Specifically, the choice of C versus a strongly typed language (such as Java) indicates that buffer overflows are a realistic threat which will be considered for test case creation and for coding efforts. Once the language is selected we can consider potential series of events which may lead to the exploitation of a vulnerability using sequence diagrams and the knowledge gained from the attack trees. For example,

Figure 4.4 shows a sequence diagram depicting the kernel stack trace when a user attempts to change the state of our driver by inputting data through a procfs virtual file. The procfs read/write routines pass a buffer from/to the kernel and as such may be a candidate for buffer overflow. The sequence diagram for the DoS caused by the concurrency problem would show a caller requesting the lock twice without releasing the lock between requests.

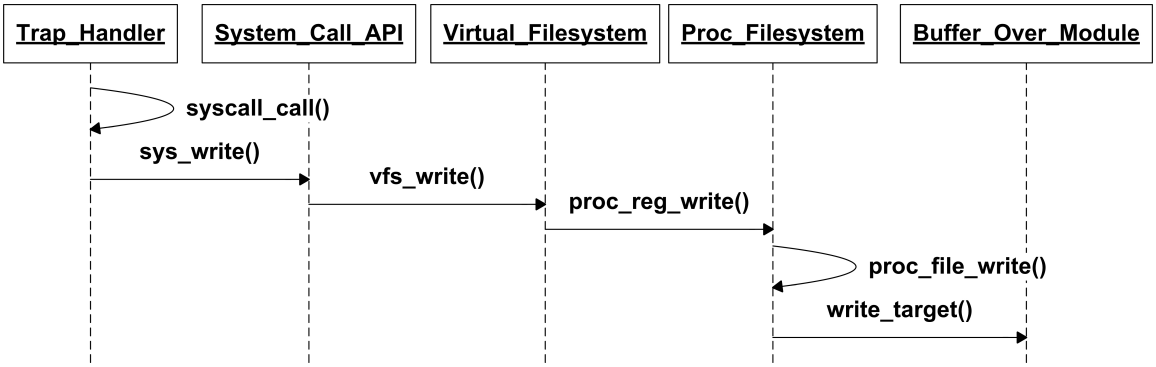

Figure 4.4: Sequence diagram depicting stack trace of call to `write_target` from the kernel perspective.

4.2.4 Contract-based Runtime Monitoring

Now that we have an understanding of the steps required to identify vulnerabilities and the contract notation that our framework employs, we will show how two vulnerabilities can be represented in LTL [118].

Specifying Security Vulnerabilities using LTL

In this subsection we show how to move from a general security vulnerability assertion in natural language to a LTL statement in preparation for the creation of our contracts. First, we will look at the example of our buffer overflow vulnerability. Second, we will explore the DoS example.

Buffer overflow is defined as the event where a source buffer with a length exceeding that of the destination buffer is placed into the destination buffer resulting in an erroneous state. Therefore, any function or method that does not perform bounds checking on buffers internally is unsafe in languages such as C and C++ and could create a potentially exploitable buffer overflow vulnerability.

For example, consider the *strcpy* function, if we let x represent the size of the source

buffer and y the size of the destination buffer, a general contract statement might be "whenever we reach a state where x=k, for some k>0, then eventually we reach a state where y==k, where the initial pre condition that x must be less than or equal to y must hold". If this assertion is violated we have shown that the attack has occurred. We now translate the statement to linear temporal logic which captures two interesting properties for security. First, a *liveness* property states that something good continues to happen. Second, a *safety* property states that something bad never happens. These provide an excellent foundation for security assertions for our contracts. The modal operators, used below, in temporal logic for *always* and *eventually* are represented symbolically as $\Box$ and $\Diamond$, respectively. The above statement can therefore be written in first-order temporal logic where the second condition represents a safety property (since having a destination that is smaller than the source will lead to issues):

$$\Box(x > 0 \rightarrow \exists\, k.(k = x \wedge \Diamond y = k)) \wedge \Box(x <= y)$$

Denial of service is defined as being a long-term inhibition of service. Delay of service is a lesser attack where the user has a temporary inhibition of service.

Potential inhibition of resources or services are a concern for most systems being designed. A long-term inhibition of service, resulting from a direct attack on a system, is called a denial of service (DoS) attack. There are many different types of DoS attacks that can be executed including protocol, logic, and bandwidth attacks. Logic attacks exploit vulnerabilities in software and are a prime candidate for monitoring.

Two different approaches to detecting an inhibition of resources include monitoring the requests for the resource (such as the request for a concurrency primitive) or monitoring the different requesters using a sampling approach.

For the first case, consider the concurrency issue when a task enters into a self-deadlock, where a task requests a locking primitive that it already holds (possible for semaphores and spinlocks). We can express this as "whenever we reach a state where we hold the lock for resource x, and we request the lock for resource x a second time, then eventually we have a self deadlock DoS". A simplified form of this statement can be written by letting state p represent the state where a task already holds the lock and is requesting the same lock a second time, and state q be the deadlock state. Thus we state the safety property that we should never be in a self-deadlock state.

$$\Box\neg(p \rightarrow q)$$

For the second case, consider availability requirements that are not met because of an intentional attack preventing those requirements from being met. In general we could express this as "whenever we reach a state where an availability requirement x is violated for some resource y then eventually we have a DoS". For example, if we let x be the cpu consumption of a particular process and y be the threshold which it cannot pass we have the following: "whenever we reach a state where x consumes more than $y\%$ of a processor for more than 10 seconds, where the initial pre condition that x must be less than or equal to $y\%$ for a given processor over a period of 10 seconds, then eventually we have a DoS". If this assertion is violated we have shown that the attack has occurred.

Now that we have shown several LTL expressions for vulnerabilities we will explore how they can be translated into contracts that can be monitored during runtime.

Verification of Buffer Overflow by Contract

Once software implementation has begun we can factor in contract-based security assertion monitoring. Potential buffer overflows can be detected using approaches such as static analysis or code review as shown at the beginning of this section. Showing whether or not a potential overflow is exploitable can be accomplished formally or empirically. For the second case we can use monitoring to informally verify that such an attack is possible. Using the temporal operators for *sometime* and *always* the contract can be written to be evaluated when monitoring completes. When monitoring completes, and there are no more states, the contract is either satisfied or not. If a rule is not satisfied then the contract will evaluate to false if the rule was specified as a liveness property (min) and true if a safety property (max) due to the semantics of safety and liveness discussed before. Keep in mind, however, that we will not likely have sufficient test coverage for safety properties to claim that the vulnerability is not possible. We can only state that it was not vulnerable given a set of input vectors and environmental conditions. Verification of a buffer overflow using the temporal logic statement above will require the following metadata:

1. Target function for breakpoint
2. Name or size of buffer argument being passed to function
3. Size of the destination buffer

Using these properties we can then realize the LTL statement, from above, with the following contract that is intended to log an event to a *buffer_log* if the destination buffer is overflowed in the target function `strcpy`.

$$E = \underline{\log}\, buffer\_log$$
$$\underline{\min}\, R(int\ k) = Sometime(y = k)$$
$$\underline{\text{mon}}\, M = Always(x > 0 \rightarrow R(x) \wedge x \leq y)$$
$$C = strcpy\ M\ E$$

To apply a contract for buffer overflow against part of our conceptual device driver we would need to find a function with a potential buffer overflow that is similar to `strcpy`. Recall that `write_target` is such a function that could have been identified either through code inspection or static analysis. The body of `write_target` references a global variable named *bad_string* that has been declared to have a fixed length of 27.

```
1   /* Expect integer <= nine digits followed by a '\n' */
2   static int write_target(struct file *file,
3       const char *buffer, unsigned long count, void *data) {
4       /* This code should be here to be secure */
5       //if (count > sizeof(bad_string))
6       // return -EINVAL;
7       if (copy_from_user(bad_string, buffer, count))
8           return -EFAULT;
9       return count;
10  }
```

Notice that the conditional that checks to ensure that the incoming buffer (char *buffer) is smaller than the destination buffer (char *bad_string) is commented out. Then the incoming buffer is copied over the destination buffer without ensuring that it will not overflow the destination. While reading from the buffer will not cause any problems, a write operation has a high possibility of causing a buffer overflow. To exploit this overflow vulnerability we could execute the following command that writes a buffer, with greater than 27 characters, into a virtual file which provides an interface into the driver eventually calling the function `write_target`.

```
# echo -n "01234567890123456789012345678901234567890123456789012345678901234567890" >
    /proc/target
```

When the system executes the above command the operating system could enter a hung state (a form of denial of service), continue for a period of time before exhibiting abnormal behaviour, or cause some form of elevation of privilege (as is often the case with stack smashing user-space buffer overflow attacks). The result is dependent upon the contents of memory that are overwritten and the code which uses the overwritten memory. Such behaviour should be disallowed regardless of the potential outcome.

For our example, we have manually written the sample code and matching probe to show the feasibility of using probes as a security monitoring mechanism. We now need to load the driver and obtain the address of the symbol and the address of the destination buffer so that we can create the probe.

```
# insmod buffer_over.ko
# grep write_target /proc/kallsyms
e0930000 t write_target [buffer_over]
# grep bad_string /proc/kallsyms
e09305e4 d bad_string [buffer_over]
```

The first command loads the driver into the kernel while the grep commands are retrieving the address of the function and the buffer from the kernel symbol table from the text (t) and data (d) sections, respectively. These will be passed into the probe to satisfy the requirements of the contract. With this information we then load our probe into the kernel to monitor the suspected vulnerability as follows:

```
# insmod catch_buffer_probe.ko breakpoint=0xe0930000 buffer_addr=0xe09305e4
```

Once the probe is in place, implemented as a jprobe (discussed later), we have the ability to monitor all calls to `write_target` for the potential occurrence of an incoming buffer that is larger than the target *bad_buffer* using the following logic:

```
1  int j_write_target(struct file *file, const char *buffer,
2  unsigned long count, void *data)
3  {
4      int len = 0;
5      ...
6      len = strlen(bad_buffer);
7      printk("The length of the destination buffer is: %d\n", len);
8      if (count > len) {
9          /* Security Violation Reaction Here */
```

```
10          printk("VIOLATION!!!\n");
11      }
12      jprobe_return();
13      return 0; /*NOTREACHED*/
14  }
15  ...
16  /*jprobe*/
17  static struct jprobe my_jprobe = {
18      .entry = (kprobe_opcode_t *) j_write_target
19  };
```

If an overflow is attempted the probe will log a VIOLATION to indicate that the assertion has been violated. Thus during execution we can start monitoring for potential violations. If we were then to rerun the command to overflow the buffer as before we would observe the following log event.

```
...
JPROBE_FUNCTION
The value of the incoming buffer is: 012345678901234567890123456
The lenth of the incoming buffer is: 27
The length of the target buffer is: 26
VIOLATION!!!
```

Verification of DoS by Contract

Rather than having a bug directly related to the code in the driver, it is possible to have a vulnerability that exists in an API or in the linkage between a component and an API. The following DoS example has to do with the order in which concurrency routines are invoked. The monitoring logic here requires two contracts. The first contract is related to the checking out of a spin lock while the second is related to the return of a spin lock.

$$E_1 = \underline{\text{resp}}\ reboot$$
$$E_2 = \underline{\text{dstruct}}\ (spinlock\_t * lock\_holder = current)$$
$$E_3 = \underline{\text{dstruct}}\ (spinlock\_t * lock\_holder = null)$$
$$\underline{\text{mon}}\ M_1 = Always((spin\_islocked(lock) \wedge lock\_holder == current) \rightarrow Self\_Deadlock)$$
$$\underline{\text{mon}}\ M_2 = (lock == target\_lock)$$

$$C_1 = spin\_lock\_irqsave\ M_1\ E_1\ M_2\ E_2$$
$$C_2 = spin\_unlock\_irqrestore\ M_2\ E_3$$

Similar to the buffer overflow example in the previous section, the DoS vulnerability is also exploited through the procfs file descriptors when a user is configuring the device. In this case, one of the read or write functions in the driver which is called when the file descriptor is used makes a call to `do_really_important_work`.

```
1  /* Routine which requires lock */
2  static void do_really_important_work(void) {
3      unsigned long flags;
4      spin_lock_irqsave(&bad_lock, flags);
5      /*Critical Section*/
6  ...
7      other_important_work();
8  ...
9      /*End Critical Section*/
10     spin_unlock_irqrestore(&bad_lock, flags);
11 }
```

Within the critical section of this function, protected by spinlock *bad_lock*, a call is made to `other_important_work`. This function may or may not be part of the driver; however, if the function also has a critical section protected by the same spinlock the system will enter into a self-deadlock.

```
1  /* Routine which requires lock */
2  static void other_important_work(void) {
3      unsigned long flags;
4      spin_lock_irqsave(&bad_lock, flags);
5      /*Critical Section*/
6  ...
7      //Code never reached if bad_lock was held previously
8  ...
9      /*End Critical Section*/
10     spin_unlock_irqrestore(&bad_lock, flags);
11 }
```

The deadlock, which occurs when a spinlock is requested a second time by a task already holding it, is extremely costly to the system. This DoS will cause an

entire processor to busy-wait indefinitely, resulting in a completely hung uni-processor machine or a severely restricted multi-core machine.

To protect against such a vulnerability being exploited our probe will have to keep track of the address of the suspect lock and the task holding it.

```
1   spinlock_t *badlock;                /*badlock address*/
2   struct task_struct *lock_holder;    /*task holding badlock*/
```

JProbes are used to apply our contract to two routines in order to capture the vulnerability (`spin_lock_irqsave` and `spin_lock_irqrestore`).

```
1   unsigned long __lockfunc j_spin_lock_irqsave (spinlock_t *lock) {
2   ...
3       if (spin_is_locked(lock)) {
4   ...
5           if (lock_holder == current) {
6               printk("Self-deadlock detected\n");
7               emergency_restart();
8           }
9       }
10      if (lock == badlock) {
11          lock_holder = current;
12          printk("bad_lock held by %s\n", current->comm);
13      }
14  ...
15      jprobe_return();
16      /*NOTREACHED*/
17      return 0;
18  }
```

To complete the logic of the probe we also need to clear the *lock_holder* when *bad_lock* is released.

```
1   void __lockfunc j_spin_unlock_irqrestore (spinlock_t *lock, unsigned long flags) {
2   ...
3       if (lock == badlock) {
4           lock_holder = NULL;
5           printk("bad_lock released by %s\n", current->comm);
6       }
7   ...
8       jprobe_return();
```

```
9         /*NOTREACHED*/
10 }
```

Once the driver is installed, and the probes put in place, we can validate the existence of the vulnerability by invoking the read routine associated with the resource requesting the spinlock. The resulting reboot of the system refreshes the kernel ring buffer; however, if we were to use `kernel_restart` (which is not interrupt-safe) instead of `emergency_restart` we could see printk messages from the syslog daemon output in */var/log/messages*.

```
Feb 10 08:24:34 localhost kernel: bad_lock=0xe0a6f500
Feb 10 08:24:34 localhost kernel: jprobes registered
Feb 10 08:24:43 localhost kernel: bad_lock held by cat
Feb 10 08:24:43 localhost kernel: Self-deadlock detected
```

The system is rebooted as soon as the self-deadlock is detected and the system returns to a stable state. This concludes our verification of the buffer overflow and DoS vulnerabilities using contract based assertions.

4.3 Realization of Contracts

Initial experiments were necessary to show that contracts could be applied to verify the presence of a vulnerability even at the lowest levels of a software architecture. We have chosen the Linux operating system for our prototype and our experiments since Linux is used for servers, super computers, embedded devices, and desktops throughout the industry. Once a potential vulnerability has been identified we create a contract that states the requirements for a secure system in which the vulnerability cannot be exploited. Then from the contracts we generate assertion-based probes that will be used during execution to monitor the system. During execution the probes can feed information back into the monitoring framework for further analysis.

Realization of such a system requires a mechanism to integrate the monitoring probes into existing systems. While some assertion based systems require the modification of existing code and compile time options [58], we have chosen an approach that does not require a change to the existing code. Modification of code can result in a runtime behaviour that is different than the unmodified version, having undesirable

consequences for analysis. The use of software breakpoints allows for a lightweight approach(depending on the logic in the probe), allowing us to insert new code into a running system(in the form of a probe), without affecting the normal behaviour of the code. Both approaches would affect the timing of the execution; however, the breakpoint approach does not cause addresses to be shifted in the machine code.

In software development, a breakpoint is typically a stopping or pausing point in execution when a developer has the opportunity to inspect and change the current context of execution. Probes on the other hand, are instruments or mechanisms used to investigate and discover properties of an unknown element.

Since the 2.6.9 Linux kernel, developers have had the use of Kprobes to insert code dynamically into the kernel using kernel modules. The Kprobes kernel patch for Linux, developed by the Linux Technology Center (LTC) at IBM, provides an extension of the processors breakpoint mechanism allowing for the integration of new code into the running kernel. The framework provides three primary mechanisms as follows:

1. ***Kprobes***: integrate new functionality before or after an executable statement, or add fault handling code in the case where a fault arises during execution
2. ***Jprobes***: access arguments passed to an executable statement, providing the ability to examine or override the default functionality of the statement
3. ***Returnprobes***: examine or override the return value of an executable statement

We incorporate these mechanisms into the design of our model, as can be seen in Figure 4.5, to realize our contracts. A security expert would be able to create the contract-enforcing probe in user-space, where they have the rest of their applications and libraries, and then use necessary commands to insert the probe into the kernel-mode memory to enforce the contract. Having the probe enforced in kernel-mode, the security contract has access to all the necessary layers in the software stack since it will have full access to the instruction set of the processor by operating in ring-0.

As discussed in Chapter 3, security vulnerabilities can be identified using a variety of methods including static analysis, penetration testing, fuzzing, and code review. The identification of a vulnerability should have with it an associated notion of confidence that should represent whether or not the potential vulnerability is actually exploitable by a malicious attacker. The purpose of our work is to assist in the runtime verification of vulnerabilities thus reducing cost and security risk by attempting to verify at least a subset of potential vulnerabilities. Verified vulnerabilities are those which must be

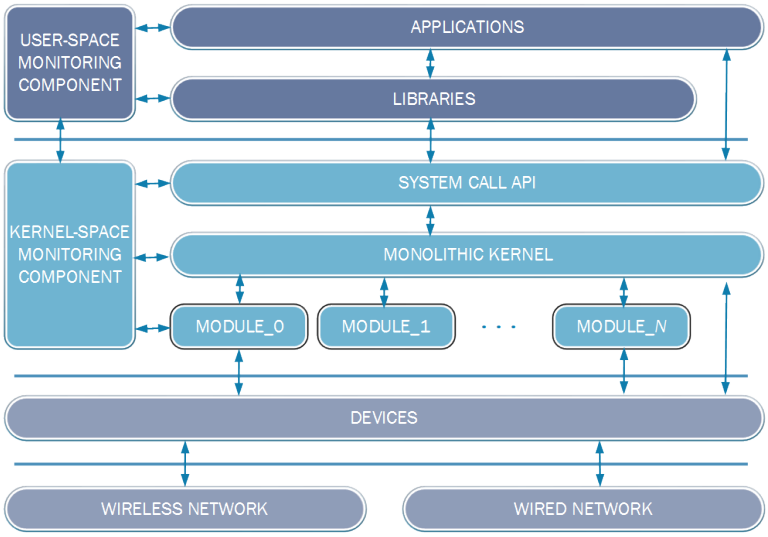

Figure 4.5: High level composition of monitoring framework.

rectified; however, the remaining unverified vulnerabilities will require further analysis (see Figure 4.6, a sub-portion of Figure 3.1).

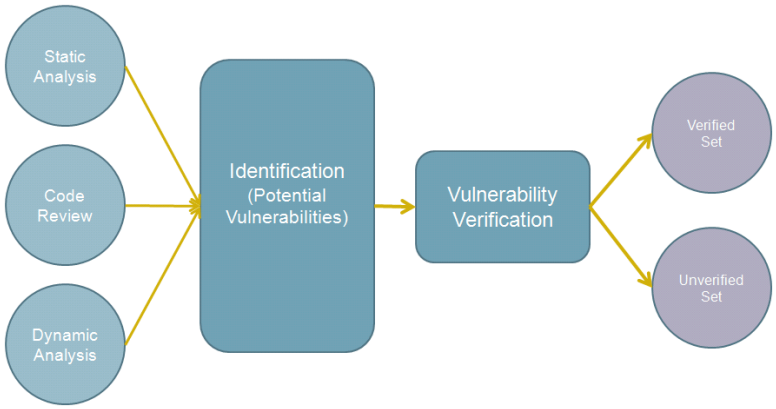

Figure 4.6: Method of deriving contracts from identified vulnerabilities.

During the Software Development Life Cycle (SDLC) different artifacts demand various approaches for the identification of potential vulnerabilities (such as the process we proposed in Figure 3.1). Figure 4.7 depicts our workflow that is used for identifying and verifying potential vulnerabilities. For example, suppose we have a large project written in the C programming language and we have identified that static analysis is an appropriate first vulnerability identification method. We would then review the results and prioritize the vulnerabilities according to our security policy documentation and project requirements. Next, contract-based security assertion monitoring framework (CB_SAMF) contracts are created using automated, semi-automated, or

manual approaches. Once the contracts are created runtime probes are generated and inserted into the target software and monitored for violations of the security assertions specified in the contracts.

4.4 Summary

This chapter introduced the syntax and semantics of our proposed security contract model along with a case study depicting how both buffer overflow and race conditions could be verified at the kernel layer of the Linux operating system. In the following chapter we present a series of existing metrics that can be used to effectively measure software security assurance approaches before introducing several new metrics for improving security assurance programs.

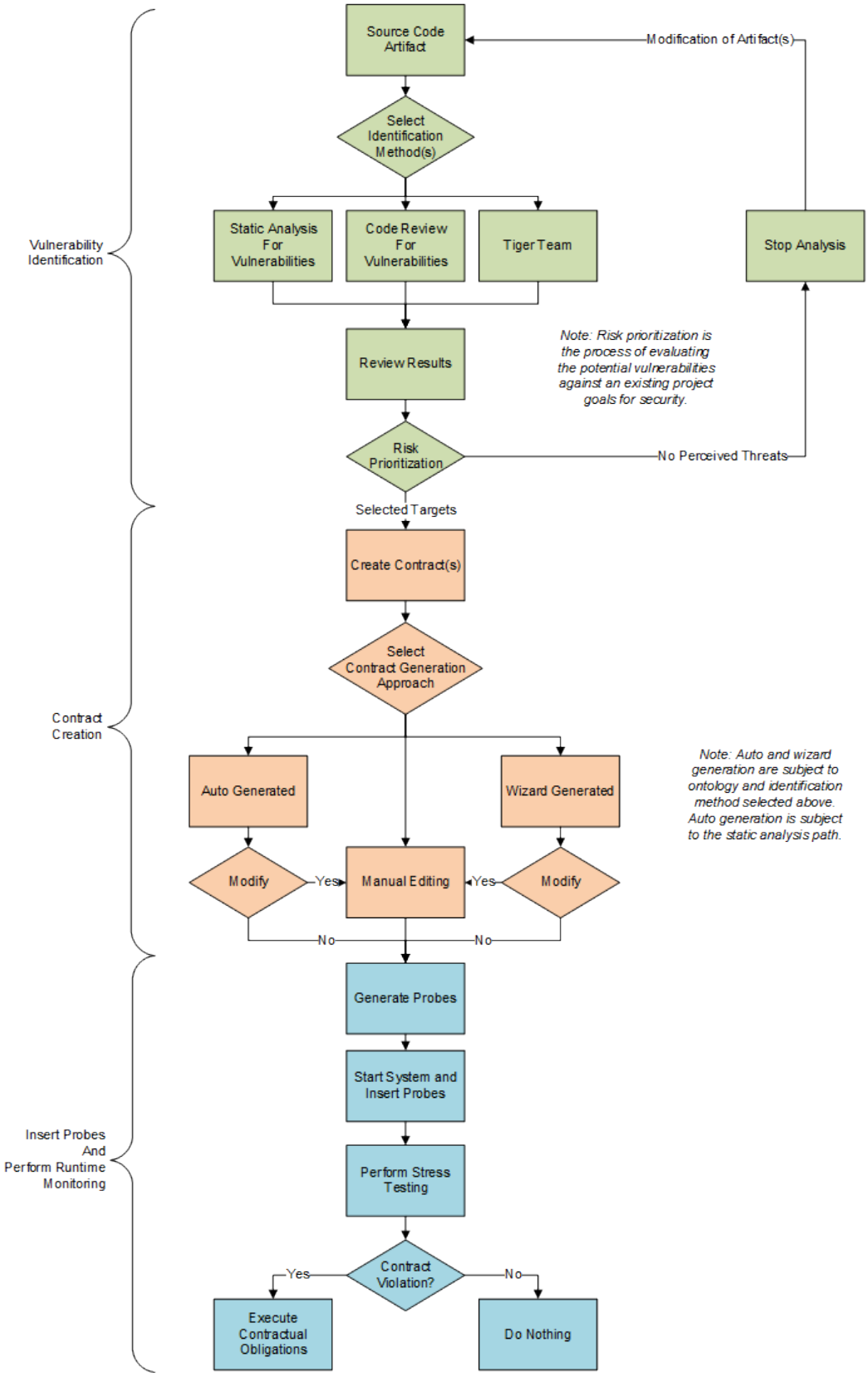

Figure 4.7: Workflow of activities involved in CB_SAMF.

Chapter 5

Metrics for Assessing and Improving Security Assurance

One accurate measurement is worth a thousand expert opinions.
-Grace Murray Hopper
We must make the important measurable, not the measurable important.
-Robert McNamara

Security requirements are fundamentally different from other software requirements. Consequently, typical quality assurance (QA) tasks that are conducted for software development do not suffice. For example, obtaining 100% branch coverage and 100% code coverage are often sufficient for functional testing. Looking closer reveals that 100% code, or branch, coverage is stating a property regarding reachability (in the sense that each statement can be reached and each branch exercised). Testing for security weaknesses requires further testing since many coding mistakes related to security are syntactically correct and are reachable. Security testing is more about determining underlying properties of the system that have negative security implications. Furthermore, being able to prioritize the order in which security weaknesses are addressed is paramount to reducing risk as developer resources are always in demand. Such prioritization can be done objectively using security metrics. Security metrics are needed in evaluating objectively the efficiency and the effectiveness of software security assurance programs and products.

While some metrics have already been shown to be useful for comparing security

assurance products, they can be applied in ways that provide misleading conclusions. During our literature review, we identified several applications of metrics that could lead to misinterpretation of results. To ensure that the way we measure the effectiveness and efficacy of our CB_SAMF we describe in detail both the background and meaning of the metrics that already exist and the ones we introduce. In essence, any mechanism that is intended to identify flaws in a system can be reduced to a binary classification problem in the sense that an approach will assert that a reported flaw either exists or does not. In order to measure the correctness of an approach we require both measurement strategies and test suites that can verify the accuracy of the reported flaws. This chapter reviews existing metrics for assessing a security assurance product and introduces new metrics for improving security assurance programs over time.

5.1 Metrics for Security Assurance Products

The assessment of the effectiveness of security assurance products and programs is based generically on the same metrics used in pattern recognition. In addition to the terms defined in Section 3.1.1 Black *et. al.* define the following relevant terms in [106]:

> ***false negative* (FN)**: When a tool does not report a weakness where one is present. [...*omitted*...]
>
> ***true positive* (TP)**: When a tool reports a weakness where one is present.
>
> ***false positive* (FP)**: When a tool reports a weakness where no weakness is present.
>
> ***false positive rate***: The number of false positives divided by the sum of the number of false positives and the number of true positives.

As such, every security vulnerability depends upon the existence of one or more security weaknesses within a target software artifact. These weaknesses create opportunities for an attacker to cause the software system to behave in an unintended fashion resulting in the access/modification of important data, interruption of normal execution, performing of actions outside of the allowed sphere of permissions, or violation of any number of additional system requirements. The terms above, combined with *true negative* (TN) which specifies a tool correctly does not report a non-existent weakness, combine to form a contingency table that will later be used as a basis of comparison (see Table 5.1). From the table, the columns for Actual Weakness and

Non-Existent Weakness are represented in later experiments by validated test suites TS100 and TS101, respectively. The rows for *Reported Weakness* and *Unreported Weakness* represent what is reported by a particular analysis method when attempting to identify weaknesses. Later, we will use these middle two rows as input to traditional metrics in order to identify potential deficiencies in security assurance tools.

	Truth		
	Actual Weakness	**Non-Existent Weakness**	
Reported Weakness	TP	FP	Total Reported
Unreported Weakness	FN	TN	Total Not Reported
	Total Weaknesses	Total Non-Existent Weaknesses	

Table 5.1: Weakness identification contingency table.

We agree with the overall definition for terms given by Black *et al.*, with the exception of *false negative* which had a second sentence "If the tool does not claim to identify a certain class of weakness, not reporting a weakness of that class is not a false negative" [106]. We omit the second sentence of the definition for *false negative* since the goal of these tools is to report all relevant security vulnerabilities (i.e. satisfy mandatory requirement SCSA-RM-1). Other areas of science such as medicine classify a false negative, or type II error, as a result that is erroneously classified in a negative category because of imperfect methods or procedures. If a security weakness is present and the analysis does not report it, then the lack of reporting gives rise to a false negative (just as the reporting of a security weakness that does not exist gives rise to a false positive). The definition of a false negative remains true whether or not a specific product is designed to detect a particular weakness type. Supporting a given weakness type also does not infer coverage of that weakness type for every permutation (i.e. occurrence of weakness in every API for a given programming language), therefore, the definition of "claim support" or "designed to detect" becomes very subjective. This is especially true of certain types of weaknesses that are tightly coupled to the APIs being used by the program. Cross-Site Scripting (XSS), for example, is specifically identified by PCI DSS 3.0 under Requirement 6.5.7[1] as needing to be detected if

[1]XSS is prioritized by security standards and best practices including: PCI DSS 3.0 (Requirement 6.5.7), OWASP Top 10 2013 (A3), SANS Top 25 2011 (CWE-79), DISA STIG 3.7 (APP3580: CAT I), and NIST SP 800-53 R4 (SI-10).

present in the code base. If XSS is not detected by an analysis tool, and it is present, then it is a false-negative in accordance with what the software security assurance tool is intended to detect according to security policy. Ideally a given tool should be customizable to add support for additional weakness types and coverage of specific APIs in order to reduce false negatives over time.

The above terms are used to specify mandatory requirements for source code analysis tools which must be able to handle various coding complexities and specific source code weaknesses.

Several metrics are already in use for the measurement and comparison of Software Security Assurance tools. Equation (3.1), defined earlier, represents the set of weaknesses detected using a particular analysis approach for a given software artifact. This result set consists of zero or more true positives (TP) and zero or more false positives (FP). In addition, W_{ij} will not include any false negatives (FN) nor true negatives (TN). Measuring weakness detection rates as percentages will require suitable formulas. *Accuracy* [71], defined by equation (5.1), is not a suitable metric to determine the efficiency of detecting weaknesses. The results may be misleading because the total population of true negatives likely far exceeds the true positives and false negatives, that combine to form the total number of actual weaknesses. These resulting values of nearly 100% would essentially ignore the presence of false and true positives.

$$A = \frac{TP + TN}{TP + FP + FN + TN} \tag{5.1}$$

Recall, as seen in equation (5.2), is arguably one of the better metrics for comparing different approaches for detecting security weaknesses as it reports the number of correctly reported weaknesses from the total population of known weaknesses (i.e. how close are we to finding all the expected results). In order to maximize the percentage *recall*, a product will need to minimize false negatives (which reduces the risk of having latent vulnerabilities in software artifacts).

$$R = \frac{TP}{TP + FN} \tag{5.2}$$

In addition to recall, *precision* (defined by equation (5.3)) also provides a useful metric to evaluate efficiency of weakness detection methods. Precision reports the number of correctly reported weaknesses from the total population of reported weaknesses (i.e. precision provides a measure of accuracy in terms of how much noise needs to be

addressed - false positives). The primary advantage of having high *precision*, caused by reducing the number of false positives in the denominator, is that it can help reduce the amount of human intervention required to audit detected weaknesses. However, it is possible to have a high precision even though you find almost none of the expected true positives (e.g. assume there are 20 vulnerabilities to detect and a tool only finds 1 out of 20 and reports no false-positives, the precision would be 1/1+0 or 100%). Ideally, we want a way to ensure both high precision and recall while minimizing FN and FP.

$$P = \frac{TP}{TP + FP} \tag{5.3}$$

Finally, *F-Measure* (see equation (5.4)) has been widely used in natural language processing for binary classification and provides a reasonable measure of the accuracy of a given test reported as a percentage [119, 120]. *F-Measure* combines both *recall* and *precision* into a weighted harmonic mean in the form of equation (5.4), and is useful in cases where we desire the mean of a set of fractions which may contain extreme values (outliers). The α and β variables are control parameters that specify how much emphasis is placed upon precision versus recall. Equation (5.4) is then simplified into the most common form, known as the balanced form or F1 measure, where $\beta = 1$ or $\alpha = 1/2$ (see equation (5.5)). This provides equal weight between precision and recall.

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \tag{5.4}$$

$$F_1 = \frac{2PR}{P + R} \tag{5.5}$$

These traditional metrics will be used in the experimental evaluation chapter to compare results, under different assumptions, since inaccurate values for FP and TP affect measurements. The above metrics are applicable for controlled experiments where the population is well understood and expected true/false positives for the population are known.

5.2 Alternate Evaluation Metrics

During our investigation it became apparent that the industry also requires alternative metrics for evaluating different security assurance approaches and products. For

example, the evaluation in [87], compared nine different static analysis products using measurements such as false negative count, true positive count, detection percentage mean (only using categories that products claimed to cover), precision, recall, and F-measure. The challenge in conducting a comparison where subsets of the population are excluded for some products, and not for others, is that the denominators used in the measurements for each product are different, resulting in conclusions that cannot be directly compared with accuracy. It would be beneficial to compare based upon a slightly more granular approach. In essence, we should use the *arithmetic mean* per weakness category, per tool, and then use weighting and scaling to create meaningful comparisons since the arithmetic mean provides the typical value of a series of numbers specific to a given question (e.g. how well can tool X detect vulnerability category Y). It is not precise, nor accurate, to ask the question how well can a given tool detect vulnerabilities.

The above observation implies that we need an accurate way to compare products, across weakness categories, and spanning security assurance products. It would also be desirable to provide consumers with a weighted option where the stakeholder may choose which weakness categories are relevant for their particular business needs. For example, if a consumer knew that they would only be programming in C and that they would never make use of SQL databases for any of their projects, they would have little need for a security assurance product capable of detecting SQL injection weaknesses. Thus any metric that provided strong scoring in SQL injection would be somewhat irrelevant.

5.2.1 Default: Arithmetic Mean

The arithmetic mean, defined as:

$$A = \frac{1}{n} \sum_{i=1}^{n} a_i \tag{5.6}$$

is used to find the *TP Rate* (or Sensitivity or Recall from Equation (5.2)) for each weakness category, where the test suite already knows how many true positives exist within the population of size n (i.e. the number of *TPs* and *FNs* sum to form n - the number of test cases expected to contain an exploitable weakness in the given category). Similarly, it is also used to find the *FP Rate* where the test suite already

knows how many false positives could exist within the population (i.e. the number of *FPs* and *TNs* sum to form n).

In the above formula, consider the case where we have 10 programs that have each been seeded with exactly one exploitable *OS Command Injection* weakness. If a software security assurance tool were able to detect eight of the 10 weaknesses correctly, then we would compute the true positive rate (TPR) by letting a_i evaluate to 1 when the tool successfully detects *OS Command Injection* in program i (0 otherwise).

$$TPR = \frac{1}{10}\sum_{i=1}^{10} a_i = \frac{8}{10} = 80\% \tag{5.7}$$

5.2.2 Artificial Scaling

There is an inherent risk in using scaling when comparing heterogeneous items in a test suite. For example, if a test suite had 10 programs that each had one intentionally exploitable *Stack Overflow* weakness, and there were five additional weakness categories in the test suite with 4 vulnerable programs each, then the greatest common denominator across all weakness categories is the 10 unique vulnerable test cases for *Stack Overflow*. If we scale all other weakness categories to have the same denominator we would give all categories the same weight (for the purpose of computing arithmetic mean for a particular tool's effectiveness). This is the implied evaluation conducted in [87], where the *Detection Percentage Mean* computes the mean of the percentages across 22 categories.

The fundamental issue that occurs with this artificial scaling is that certain assumptions do not hold for security assurance products. There are at least three confounding factors with such an approach:

1. **Coding constructs:** each test case in a test suite, for a given weakness type, may employ different coding constructs that lead to the vulnerability (e.g. loops, conditionals, inter-procedural data flows, pointer arithmetic, and others). It is not valid to assume that if an analysis product handles one coding construct, then it can handle all coding constructs.
2. **Heterogeneous APIs:** different function calls will be employed in different combinations for various test cases for a given weakness type. It is not valid to assume that a product which can detect a weakness (of a given type) when a particular

function is called, then it can handle all other functions in different APIs (which lead to the same weakness type).

3. **Relevant importance:** different software products will make use of different programming frameworks and APIs within a given programming language. As a result different weakness categories will become more, or less, relevant to a particular project. It is not valid to assume that all stakeholders will consider all weakness categories relevant for their particular programming project(s).

When each weakness type in a test suite contains different permutations of the above factors, such as not having the same number of test cases, handle the same number of constructs or APIs, nor have the same relative importance, then scaling should not be used to compute metrics used for comparison. We should rely on direct comparison using arithmetic mean or alternatively a useful weighted mean.

5.2.3 Consumer: Weighted Mean

The weighted mean, of non-negative weights, for a non-empty set of data (e.g. test cases - per weakness category):

$$x_1, x_2, ..., x_n$$

is defined as:

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \tag{5.8}$$

When the weighted arithmetic mean is adjusted in order for some weakness categories to be eliminated, such that they are no longer considered as part of the mean, we can customize measurements for a stakeholder who is only interested in particular weakness types. This weighting must be applied consistently across all tools if the desire is to compare efficacy of tools.

For example, envision a stakeholder who only considers the following five weakness categories: {Format String Vulnerability, Resource Injection/Path Manipulation, OS Command Injection, SQL Injection, and Basic XSS}. Consequently, the weight of '0' can be applied for all other categories in a test suite they are not interested in. For the five targeted categories, a weight of '1' is applied to keep their default weighting.

We assert that a framework permitting stakeholder-defined weighting would be

useful for conducting evaluations.

Finally, further work in designing useful test suites for tool comparison is still needed to handle the complexities mentioned above relating to coding constructs, heterogeneous APIs, and relevant priorities. Ideally, effort should be exerted to create sufficient test suites so that artificial scaling would never be considered. Assuming that test suites exist in the future that provide sufficient coverage, stakeholders would then be able to not only use inclusion/exclusion weighting but also relative weighting to reflect how important certain weakness categories are to their particular application security program (i.e. stakeholders could adjust weighting such that SQL injection was half as important as XSS).

5.2.4 Verification Metrics

Using data collected during an updated SDLC, stakeholders will be able to answer several additional questions if we introduce several new metrics. These metrics are related to verification of vulnerabilities during development and testing, as part of the process we introduced in Figure 3.1, and our confidence in their exploitability. Such questions would eventually include the following:

- Which vulnerabilities can be verified automatically?
- Which vulnerabilities can be detected automatically?
- Which vulnerabilities can absence be verified?
- Which vulnerabilities do we need to look for during code review?
- Which vulnerabilities should we fix first?

In order for us to answer the above questions for a single source artifact s_j, from Figure 3.1, we need to define the superset of all weaknesses (i.e. suspected vulnerabilities), that could be detected by all mechanisms x_i for a source artifact.

$$W_j = \bigcup_{i=1}^{n} W_{ij} \tag{5.9}$$

Furthermore, we also require the superset of all possible weaknesses for all source artifacts in order to answer these questions in general.

$$W = \bigcup_{j=1}^{m} W_j \tag{5.10}$$

If we let W be the set of all weaknesses which consist of $w_1, w_2, ..., w_n$, and we use the variable l to iterate across them, then additional metrics to track data related to the above questions would include the following:

- **Vulnerability Confidence Score (VCS)** is a measurement over time of how confident we are that a suspected vulnerability (i.e. detected weakness) is an exploitable vulnerability. It is the total number of times unique instances of the vulnerability type have been verified using runtime monitoring (which have been reported by a weakness detection method) over the total number of times unique instances of the vulnerability have been suspected (using any of the detection mechanisms x_i in Figure 3.1). This metric is maintained for each vulnerability type. The purpose of this metric is to provide an indicator as to how likely it is that a suspected vulnerability is reachable and exploitable. The calculation for this metric is as follows:

$$VCS_{category} = \frac{\sum_{l=0}^{l=n} vulnerability\_verified(w_l)}{\sum_{l=0}^{l=n} vulnerability\_suspected(w_l)}$$

  Assuming that eventually we reach the state where we have 100% of vulnerabilities verified, this is akin to *TP+FN/TP+FP* across vulnerability detection mechanisms. When all detected vulnerabilities are verified and all vulnerabilities are detected, without any false positives, this simplifies to TP/TP since there would be no FNs nor FPs (thus VCS=100%). If there are vulnerabilities that were not detected, but no FPs, we would have TP+FN/TP. If there were vulnerabilities that were incorrectly identified, but no FNs, we would have TP/TP+FP. Ultimately, it is much more likely to have both FPs and FNs with a given approach within a security assurance program. Finally, use of VCS must be completed on a per category basis and be updated as new 0-day vulnerabilities are disclosed and tool coverage is expanded. For example, when a new vulnerability type is created there is no VCS measurement for it and when tools begin to detect the new type, the number of detected vulnerabilities will likely grow before verification is even possible.

- **Vulnerability Detectability Score (VDS)** is a measurement over time of how confident we are that a vulnerability type can be successfully detected. VDS is maintained for each vulnerability type. It is the total number of verified

vulnerabilities over the total number which exist (new vulnerabilities, which may have been verified by runtime monitoring, but not yet detected by other tools). The purpose of this metric is to provide an indicator as to how easily a particular vulnerability type may be accurately detected using the tools in a security assurance program. The calculation for this metric is as follows:

$$VDS_{category} = \frac{\sum_{l=0}^{l=n} vulnerability\_verified(w_l)}{\sum_{l=0}^{l=n} vulnerability\_exists(w_l)}$$

Assuming that eventually we reach the state where we have 100% of vulnerabilities verified, this is akin to *TP/TP+FN* across vulnerability detection mechanisms. Finally, the use of VDS in practice requires data feeds which accurately categorize vulnerabilities to their correct type and users must be cognizant of the fact that a single tool may not be capable of finding a particular category. Thus, both VCS and VDS should be maintained per category across multiple detection mechanisms.

Tracking VCS per vulnerability category would keep track of all of the weaknesses detected using various mechanisms (such as static analysis, dynamic analysis, and code review) and which of these are verified to be exploitable vulnerabilities.

Maintaining VDS for each vulnerability type would reflect both those instances of vulnerabilities which were detected and verified (i.e. true positives) and also those weaknesses which were not detected and were later exploited (i.e. false negatives).

Stakeholders who track the above metrics relative to their own application security program would be able to make more accurate decisions based upon additional risk data when a score is not near to 100%. For example, if the stakeholder knew that the VCS for *OS Command Injection*, which is also a critical weakness type, was near 100% then they would know that they should always remediate these weaknesses when they are detected. In addition, if the stakeholder also had high confidence that *OS Command Injection* was almost always detectable using their security assurance program (based upon their tracked VDS) then they would not expend additional resources on code review to identify instances of the weakness category.

Ultimately, such metrics would also assist stakeholders identify deficiencies in their current application security assurance programs to make further effectiveness

improvements. Low VCS values for particular categories would indicate the need to make improvements for detecting noted categories to reduce false positive noise. On the other hand, a low VDS value for a given vulnerability type would indicate improvements are needed to simply detect the risk (reduce false negatives). Later, in our evaluation, we will present experimental results that show how low VCS and VDS for several categories could lead to improvements in an application security program.

5.3 Summary

This chapter reviewed several metrics commonly used in pattern recognition, discussed how those metrics can be used in the evaluation of security assurance products, introduced alternative evaluation metrics (along with arguments on how they should not be applied), and introduced two new metrics for measurably improving a security assurance program which uses tools/products to reduce security risk.

Next, in order to evaluate the verification of vulnerabilities using CB_SAMF we have designed a series of experiments to evaluate the effectiveness of runtime monitoring of suspected vulnerabilities. For this evaluation we have selected several datasets and use one of the best static analysis tool sets on the market to identify potential vulnerabilities, from which we then create CB_SAMF contracts for runtime verification. During these experiments, we will apply the above metrics to evaluate the efficiency of our monitoring approach before applying metrics to improve a security assurance program which uses a static analysis product.

Chapter 6

Experimental Evaluation

False facts are highly injurious to the progress of science, for they often long endure; but false views, if supported by some evidence, do little harm, as every one takes a salutary pleasure in proving their falseness; and when this is done, one path towards error is closed and the road to truth is often at the same time opened.
-Charles Darwin

In this chapter, we conduct a series of experiments to assess the effectiveness of our contract-based monitoring framework and the new security assurance metrics introduced in the previous chapters. This chapter is broken into two primary parts targeting experiments for runtime monitoring and applicability of security metrics, respectively. First, Section 1 introduces the experimental context, taxonomies, and test suites while Section 2 discusses the two selected vulnerability datasets. Section 3 covers *Part I* of our experiments which focuses on evaluating the effectiveness of our proposed model for verifying that suspected vulnerabilities are exploitable given their underlying weaknesses using monitoring, as well as the empirical evaluation results that led to the identification of test suite deficiencies and the creation of replacement test suites. Section 4 covers *Part II* of our experiments, related to security metrics, where we apply metrics from Chapter 5 to show how deficiencies in test suites lead to incorrect evaluations, and how security metrics can be used to improve software security assurance practices.

6.1 Experimental Context

Software targets for runtime monitoring can span multiple software layers. The case study in Section 4.2 showed the efficiency of monitoring device drivers within the kernel layer. For the purpose of showing the efficacy of our model, our focus in this chapter will be to expand our monitoring evaluation to include user-space binaries within the Linux operating system. Specifically, we are targeting executable and linking format (ELF) binaries compiled from C source file applications. We will describe the host where these experiments were conducted, discuss related critical weaknesses (introduced in Section 3.5) within the context of known weakness and vulnerability taxonomies, and identify relevant test suites for evaluating the efficacy of our approach.

6.1.1 Environment Description

Experiments were conducted using the following configuration within a virtual machine:

- RAM: 4GB virtually allocated from 16GB of physical RAM
- Hard drive: 30GB Contiguous Virtual Disk from an Intel 180GB SSD
- Processor: 1 virtual processor from an Intel i7-4600U CPU 2.7GHz (Dual Core)
- Operating System: Fedora 19
- Kernel: 3.12.5-200.fc19.x86_64
- GCC: 4.8.3 20140911 (Red Hat 4.8.3-7)
- SystemTap: 2.4/0.156, rpm 2.4-1.fc19
- Fortify Static Code Analyzer: 6.10.0120

6.1.2 Taxonomies

While there is not a universally accepted standardized taxonomy or ontology available for this research area, there are at least three projects that are working towards these goals. Wherever appropriate we will defer, for the sake of accepted terminology and classification, to the MITRE Common Weakness Enumeration (CWE)[1] project, the open source OWASP Honeycomb project[2] which became the Application Security Desk Reference (ASDR)[3], or the HPE Security Fortify Taxonomy (formally known as

[1] http://cwe.mitre.org
[2] https://www.owasp.org/index.php/Category:OWASP_Honeycomb_Project
[3] https://www.owasp.org/index.php/ASDR

the Seven Pernicious Kingdoms)[4].

6.1.3 Test Suite Datasets for Experiments

We do not propose to study the rate at which vulnerabilities are introduced or the rate at which they are removed; rather, we are concerned with how to increase the productivity of identifying and verifying vulnerabilities. Using our verification approach, defined in Figure 3.1, we propose to use functional test-case based datasets for experiments to evaluate vulnerability detection using static analysis (selected analysis method x_i). Test suites are verified using contract-based runtime monitoring to determine if the suspected vulnerabilities are reachable and exploitable. Among the many security related vulnerabilities that are to be detected and then verified are those related to OS Command Injection, Basic XSS, SQL Injection, Resource Injection, Buffer Overflow, Format String Vulnerability, Path Manipulation, Information Leakage, and basic use of Dangerous Functions.

Empirical evaluation will be conducted against a subset of the open source projects described below in Table 6.1. We eliminate the first, fourth, fifth, and sixth datasets from consideration. We omit the first dataset since we do not want to modify datasets in order to use them and the C-based Juliet test cases focus primarily on Windows-based vulnerabilities and we are focusing on Linux-based vulnerabilities. We omit the fourth dataset since it is not focused directly on detecting weaknesses but rather suppression of weaknesses. We omit the fifth dataset because it is currently unavailable online. Finally, we omit the sixth dataset because it is not publicly available.

The second and third datasets are selected for evaluation as they were specifically designed to measure the accuracy of detection techniques. The test suites behind these two datasets will be discussed in Section 6.2.

6.2 Selected Vulnerability Datasets

Improving software security assurance is not a trivial task and it requires careful thought, planning, and execution to improve the state of the industry over time. To reduce the amount of individual effort required to measure the effectiveness of a security assurance product, developers and researchers must have access to test

[4]https://vulncat.hpefod.com

Option	Name	Description	Number of Cases
1	Juliet Test Suite for C/C++ http://samate.nist.gov/SRD/testsuite.php	"This is a collection of test cases in the C/C++ language. It contains examples for 116 different CWEs. This software is not subject to copyright protection and is in the public domain. NIST assumes no responsibility whatsoever for its use by other parties, and makes no guaranties, expressed or implied, about its quality, reliability, or any other characteristic." [121]	45309 (not all usable since designed for Windows)
2	C Test Suite for Source Code Analyzer - weakness [Test Suite #45] http://samate.nist.gov/SRD/testsuite.php	"This test suite tests against Source Code Security Analyzer based on functional requirements SCA-RM-1 through SCAN-RM-5 specified in "Source Code Security Analysis Tool Functional Specification"." [121]	**75** (2 deprecated programs are included in the download, making the total number 77)
3	C Test Suite for Source Code Analyzer - false positive [Test Suite #46] http://samate.nist.gov/SRD/testsuite.php	"This test suite tests against Source Code Security Analyzer based on functional requirements SCA-RM-6 specified in "Source Code Security Analysis Tool Functional Specification"." [121]	**73**
4	C Test Suite for Source Code Analyzer - weakness suppression http://samate.nist.gov/SRD/testsuite.php	"This test suite tests against Source Code Security Analyzer based on functional requirements SCA-RO-2 specified in 'Source Code Security Analysis Tool Functional Specification'." [121]	**21**
5	Fortify OWASP Open Review Project http://owasp.fortify.com	The OWASP Open Review Project identifies and reports bugs and security vulnerabilities in widely used open source software.	76 Projects (not all C projects)
6	Open Source Security Audits https://customerportal.fortify.com/premium/openSourceAuditDocs.jsp [Requires authenticated login by a valid subscriber]	Unknown security vulnerabilities and quality bugs in open source code expose consumers of open source software to significant risk. In response, Fortify has developed an open source security initiative, called Fortify Open Review (FOR), which identifies and reports bugs and security vulnerabilities in widely used open source software and makes the results available to the open source development teams responsible for the projects.	100 Projects (not all C projects)
7	SAMATE Reference Dataset (C test cases) http://samate.nist.gov/SRD	Complete set of C test cases from SAMATE reference dataset as of October 7th 2011. This dataset includes 1165 (33 approved) Bad test cases, 434 (73 approved)Good test cases, 106 Approved Test Cases, 1493 Candidate Test Cases, and __ weaknesses.	**1599**

Table 6.1: Collection of considered datasets for empirical evaluation.

suites designed for evaluating software security assurance products. The Software Assurance Metrics and Tool Evaluation (SAMATE) project at the National Institute of Standards and Technology (NIST) has created the Software Assurance Reference Dataset (SARD) to provide users, researchers, and software security assurance tool developers with a set of known security flaws. These known security flaws take the form of applications that contain intentional security weaknesses, grouped into test suites with specific purposes, which can then be employed to measure different security attributes.

6.2.1 Challenges: Test Suites 45 and 46

NIST has accumulated many useful test suites that provide researchers and software security assurance tool developers with mechanisms to compare and improve methods for detecting security vulnerabilities. In 2006, NIST produced a pair of draft special publications (SP) specifying the minimum capabilities of a source code security analyzer against a specific set of security weaknesses. The two documents consisted of a functional specification and a related test plan that included test suites for several programming languages. The initial version was released in 2007 and after being revised, released again as version 1.1 in 2011 [106, 122].

We selected from the above test plan, test suite 45 (TS45) to test the capability of our Contract-Based Security Assertion Monitoring software security assurance prototype's handling of certain weaknesses in the C language and test suite 46 (TS46) to assess the false positive ratio [122].

6.2.2 Purpose of Test Suites

Test suites 45 and 46 specify sets of test cases, in the C-language, intended to measure the ability to successfully identify weaknesses and determine the false-positive ratio when comparing source code security analysis products.

We use *Test Suites 45*[5] and *46*[6], which were both created by Michael Koo *et al.*, based on the SCSA-RM-1 through SCSA-RM-6 requirements specified in the "Source Code Security Analysis Tool Functional Specification" as part of the NIST SAMATE project [106]. The NIST SAMATE project, beginning in 2004, has been focused on improving software assurance by developing methods to enable software tool

[5]http://samate.nist.gov/SRD/view.php?tsID=45
[6]http://samate.nist.gov/SRD/view.php?tsID=46

evaluations in support of the Department of Homeland Security's Software Assurance Tools and R&D Requirements Identification Program. The Source Code Security Analyzers effort is defined as part of a specification under NIST Special Publication 500-268 v1.1 [106] and test plan under NIST Special Publication 500-270 [122]. These two publications are intended to assist in the understanding of the capabilities of source code security analysis tools by providing a set of functional requirements for source code security analysis tools and a method for evaluation.

Koo *et al.* specify a set of metrics, including test suites and methods, to determine how well a particular source code security analysis tool conforms to the specification [122]. Test suites were provided for three example programming languages (C, C++, and Java) with the intent of evaluating the requirements and features defined in NIST Special Publication 500-268 v1.1. Each test case, within a test suite, then provides metadata including a test description, a weakness description, expected result, and source code. Test cases are atomic programs that demonstrate a given weakness (marked *bad*) in a specific code construct in order to measure a tool's ability to detect actual security vulnerabilities. Alternatively, a test case represents a fixed version of a weakness (marked *good*) in a specific code construct where the weakness has been removed to measure the capability of the tool to avoid generation of false positives. Within the context of the two selected datasets, we will refer to test cases and programs interchangeably.

6.2.3 Coverage of CWE's

Test suites 45 and 46 cover 21 specific CWE-IDs, from the Common Weakness Enumeration (CWE) types specified by MITRE[7]. These CWEs were selected as a "base set" of weaknesses spanning the category domains of input validation, range errors, API abuse, security features, time and state, code quality, and encapsulation. Test suite 45 (TS45) consists of 77 programs that are intentionally seeded with specific security weaknesses. While discussing *TS45* we should note that the NIST SP 500-270 specifies 75 test case IDs for *TS45*, while the downloaded test suite from SARD contains 77. The extra two test cases, 1446 (duplicate replaced by 2199) and 2108 (duplicate replaced by 2208), affect the test suite by artificially increasing both CWE-415 and CWE-416 by one test case each, respectively. Test suite 46 (TS46) consists of 73 programs that have had their intentionally seeded weaknesses fixed (void of weakness).

[7]https://cwe.mitre.org/

Table 6.2 provides a summary of the specific source code weaknesses that must be covered by a given source code analysis tool [106] along with an enumeration of the test cases that have been created to evaluate the detection of specific weaknesses [122]. For each program in *TS45* the expected result of running the tool is to report the associated weakness. Each program in *TS46* expects tools to not report the associated weakness category since the vulnerabilities have intentionally been "fixed". When

Category	**Source Code Weakness**	**CWE ID**	**Vulnerable Programs TS45 (SARD Test Case ID)**	**Fixed Programs TS46 (SARD Test Case ID)**
Input Validation	OS Command Injection	78	111 1881 1883 1885	2136 2137 2138 2139
	Basic XSS	80	1781 1794 1919 1921 2198	1795 1920 1922 1924 2204
	SQL Injection	89	1796 1798 1800	1797 1799 1801 1930
	Resource Injection	99	1895 1897 1899 1901	1896 1898 1900 1902
Range Errors	Stack Overflow	121	1544 1548 1563 1565 1751 1905 1907 1909 2009	1545 1547 1549 1566 1602 1906 1908 1910
	Heap Overflow	122	1611 1612 1843 1845	1574 1613 1615 1844 1848 2134
	Format String Vulnerability	134	10 92 93 1831 1833	1556 1560 1562 1832 1834
	Improper Null Termination	170	1849 1850 1854 1857 2010	1855 1856 1858 2012
API Abuse	Heap Inspection	244	1737	
	Often Misused String Management	251	1865 1867 1869 1871 1873	1866 1868 1870 1872 1874
Security Features	Hard-coded Password	259	1810 1835 1837 1839 1841	2130 2131 2132 2133
Time and State	Time-of-check Time-of-use Race Condition	367	102 1806 1808	1892 1894
	Unchecked Error Condition	391	1928	1929
Code Quality	Memory Leak	401	1585 1588	1586 1589 1925 1926 1933
	Unrestricted Critical Resource Lock	412	2109	2205
	Double Free	415	99 1827 1829 1590 2199	1591 1828 1830 2271
	Use After Free	416	2200 2201 2202 2203	1914 2135 2269 2270
	Uninitialized Variable	457	1757 2003 2019	2186
	Unintentional Pointer Scaling	468	1782	1927
	Null Dereference	476	1875 1877 1879 2193	1876 1880 2194 2195
Encapsulation	Leftover Debug Code	489	1861	1862

Table 6.2: Weakness categories covered by *TS45* and *TS46*.

the expected result for each atomic program in *TS45* is not reported, we record a false negative. When an expected result is reported, we record a true positive. For *TS46*, if the expected result is not achieved for a given atomic program, we record a false positive. To ensure the accuracy of such measurements, a detailed review of the programs and a verification that weaknesses are exploitable vulnerabilities is also warranted to ensure the sanity of results.

6.3 PART I: Verification of Test Suites 45 and 46

Building test suites or datasets for evaluating software security assurance tools is challenging due to the complexity of test subjects and the problem domain. Common sense is to rely on public datasets when adequate datasets are available. In this case, the integrity of the test results depends on the reliability of the test suites. Despite enormous efforts invested by designers of the test suites, many test sets include mistakes and oversights that potentially flaw test results. To ensure accuracy and validity of our empirical evaluation, verification of test suites is essential to ensure that they satisfy specified requirements.

We present in this section our analysis of TS45 and TS46, discuss identified deficiencies and associated improvements, evaluate how invalid assumptions lead to imprecise measurement, and introduce replacements: test suite 100 (TS100) and test suite 101 (TS101). Finally, to measure the impact of the replacement test suites when evaluating security assurance products, we compare the results of a commercial static code analysis product using *recall*, *precision*, and *F-Measure*.

Our first experiment involves conducting a code review of the programs in *TS45* and *TS46*, experimentation, and verifying inconsistent weaknesses with our proposed contract-based runtime monitoring approach. We uncovered exploitable weaknesses in 13 out of the 73 programs designed to not contain exploitable weaknesses of specific CWE types in *TS46*. Work by Díaz and Bermejo comparing various static analysis products also identifies 8 of the 13 latent defects, however, the test suite was never updated to resolve these deficiencies [87]. Thirteen out of 73 programs is significant enough to have a relatively large error that would hinder any evaluation approach comparing different approaches using statistical measures such as *false positive rate*, *recall*, *precision*, and *F-measure* [89, 119, 123, 95].

Using the approach to finding and verifying weaknesses, described in Section 3.2, we verify the assumptions of test suites 45 and 46. According to the mandatory requirements in Section 3.2.2 of the test plan, *TS46* was designed to measure the ability of a software security analysis tool to produce reasonably low numbers of false positives. Each program is meant to have a "fixed" version of a given security weakness, as can be confirmed by reviewing the metadata of each test case in SARD; however, during a manual code review and verification experiment 13 programs that were supposed false positives were still exploitable for their specific weakness type.

6.3.1 Experiment 1: Verifying Exploitability with Probes

With our dataset now selected and metrics identified our first experiment focuses on whether probes can be used to confirm the exploitation of our selected five weaknesses categories during runtime.

Population: Focus will be on programs from the following weakness categories, as presented in Section 3.5: 4 *OS Command Injection*, 5 *Basic Cross-Site Scripting*, 3 *SQL Injection*, 4 *Resource Injection/Path Manipulation*, and 5 *Format String* weaknesses. In total, we aim to verify 21 vulnerable programs, as seen in Table 6.3, as being either exploitable or not using runtime probes.

Category	**Source Code Weakness**	**CWE ID**	**Vulnerable Programs TS45 (SARD Test Case ID)**
Input Validation	OS Command Injection	78	111 1881 1883 1885
	Basic XSS	80	1781 1794 1919 1921 2198
	SQL Injection	89	1796 1798 1800
	Resource Injection	99	1895 1897 1899 1901
Range Errors	Format String Vulnerability	134	10 92 93 1831 1833

Table 6.3: Targeted programs from *TS45* that are vulnerable to targeted CWEs.

Experimental Objective and Theory: Test cases intended to be vulnerable will be verifiable using probes at runtime. The stated goal of this experiment is to verify a subset of the vulnerabilities designed in *TS45* using manually constructed runtime probes. Test cases that are not vulnerable cannot be verified with absolute confidence. Collected data includes the following: true positive verification, percentage of vulnerabilities verified, and Lines of Code (LOC). Both Executable Lines of Code (ELOC) and Total Lines of Code (TLOC) will be tracked for the code under investigation.

Results: In the unmodified Dataset 2, we were able to verify 100% of the command injection (Table 6.4), cross-site scripting (Table 6.5), SQL injection (Table 6.6), path manipulation (Table 6.7), and format string (Table 6.8). Specifically, the following tables indicate the verification that weaknesses are exploitable, for the programs (identified by Test Case ID), in each of the specified categories.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Exploitable
111	tain-bad1.c	5/33	1	☑
1881	os_cmd_local_flow.c	11/41	1	☑
1883	os_cmd_loop.c	7/34	1	☑
1885	os_cmd_scope.c	8/36	1	☑

Table 6.4: Potential for Command Injection in Dataset 2. All seeded vulnerabilities verified.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Exploitable
1781	xss_scope_bad.c	10/44	1	☑
1794	xss_basic_bad.c	10/38	1	☑
1919	xss_@alias_bad.c	9/39	1	☑
1921	xss_container_bad.c	10/44	1	☑
2198	xss_loop_bad.c	13/43	1	☑

Table 6.5: Potential for Cross-Site Scripting in Dataset 2. All seeded vulnerabilities verified.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Exploitable
1796	sql_select_bad.c	24/68	1	☑
1798	sql_array_bad.c	28/75	1	☑
1800	sql_scope_bad.c	28/77	1	☑

Table 6.6: Potential for SQL Injection in Dataset 2. All seeded vulnerabilities verified.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Exploitable
1895	resource_injection_@alias.c	10/44	1	☑
1897	resource_injection_basic.c	9/40	1	☑
1899	resource_injection_container.c	11/48	1	☑
1901	resource_injection_scope.c	10/44	1	☑

Table 6.7: Potential for Path Manipulation in Dataset 2. All seeded vulnerabilities verified.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Exploitable
10	Format_string_problem.c	3/31	1	☑
92	fmt-bad1.c	4/29	1	☑
93	fmt-bad2.c	4/29	1	☑
1831	fmt_string_local_container.c	4/32	1	☑
1833	fmt_string_local_control_flow.c	5/36	1	☑

Table 6.8: Potential for Format String in Dataset 2. All seeded vulnerabilities verified.

Conclusion: The targeted programs from *TS45* were all vulnerable, as expected, to their specified weaknesses. As such, all detections resulted in 100% TP detection. It should be noted, however, that XSS detection required a work-around since the underlying probing framework had difficulty accessing variable list arguments for functions such as *fprintf*. Furthermore, these five weakness types have similar patterns to their exploits since they require the consumption of malicious user input by vulnerable functions.

6.3.2 Experiment 2: Manual Review of Datasets

While we were able to verify all of the vulnerable test cases in Experiment 1, a counter experiment to verify that the programs for the same weaknesses categories in *TS46* were not vulnerable led to some concerning results.
A base hypothesis to be verified, is that the datasets used as the basis of an empirical evaluation are valid in accordance with their claims. While we conduct a manual review and verification of all test cases our primary focus, as per Section 3.5, we will target the following five weakness types during this analysis: Format String, Resource Injection/Path Manipulation, OS Command Injection, SQL Injection, and Basic XSS.

Population: There are 75 programs intended to be in *TS45*, each of which is to exhibit one particular weakness, spanning 21 weakness categories. Additionally, *TS46* is intended to contain 73 programs, each of which is to "NOT" exhibit a particular weakness, spanning 21 weakness categories (see Table 6.2).

Experimental Objective and Theory: Test cases intended to be vulnerable must in fact be vulnerable and test cases intended to not be vulnerable are not exploitable. The stated goal of this experiment is to review and verify the vulnerabilities designed in *TS45* and *TS46*. Collected data includes the following: code review metadata, false positive verification, true positive verification, percentage of vulnerabilities verified, and Lines of Code (LOC).

Results: All programs in ***TS45*** were vulnerable, as intended, in accordance with our verification seen for a subset of the programs in Experiment 1.

A subset of programs in ***TS46***, however, were still vulnerable in contradiction of their intended design. In the unmodified version of Dataset 3 we identified 13 exploitable vulnerabilities in applications which are not supposed to contain specific weaknesses (including 10 in our targeted weakness categories). We summarize each vulnerability type below, marking those test cases that are still exploitable with a ☒ while those that are not exploitable with a ☑.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Not Exploitable
2136	os_cmd_local_flow_good.c	19/58	0	☒
2137	os_cmd_loop_good.c	14/52	0	☒
2138	os_cmd_scope_good.c	16/54	0	☒
2139	os_cmd_injection_basic_good.c	25/60	0	☒

Table 6.9: Potential Command Injection issues in Dataset 3. All programs are incorrectly exploitable [4/4].

Four Command Injection issues in Dataset 3 should be false positives (Table 6.9). On detailed analysis, however, these all appear to be true positives as the added custom function (`purify`) that is intended to remove ";" is not sufficient to remove command injection vulnerabilities. As such any metrics related to false positive rates with this category would be completely, or 100%, inaccurate.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Not Exploitable
1795	xss_basic_good.c	11/40	0	☑
1920	xss_@alias_good.c	10/40	0	☒
1922	xss_container_good.c	11/45	0	☑
1924	xss_scope_good.c	11/45	0	☒
2204	xss_loop_good.c	15/46	0	☑

Table 6.10: Potential Cross-Site Scripting issues in Dataset 3. Two programs are incorrectly exploitable [2/5].

Two of the five programs that are intended to *NOT* be vulnerable to XSS still are, resulting in 40% of the inputs to metrics using false positives measurements for this category to be incorrect (Table 6.10).

Program	File Name	ELOC/TLOC	Intended Vulnerable	Not Exploitable
1797	sql_select_good.c	25/73	0	☑
1799	sql_array_good.c	31/74	0	☑
1801	sql_scope_good.c	29/80	0	☑
1930	sql_injection_loop_good.c	31/74	0	☑

Table 6.11: Potential SQL Injection issues in Dataset 3. All programs do not contain exploitable vulnerabilities [0/4].

All four SQL Injection test cases in Table 6.11 appear to have been corrected to avoid SQL Injection attacks through a combination of using the `mysql_real_escape_string` function to escape special characters for the character set the server is expecting, along with necessary placing of single quotes (') around the variables in the query. As a result, metrics using false positive measurements for SQL Injection would be accurate.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Not Exploitable
1896	resource_injection_@alias_good.c	14/65	0	☒
1898	resource_injection_basic_good.c	13/59	0	☒
1900	resource_injection_container_good.c	15/69	0	☒
1902	resource_injection_scope_good.c	14/66	0	☒

Table 6.12: Potential Path Manipulation issues in Dataset 3. All programs incorrectly contain exploitable vulnerabilities [4/4].

These four programs in Table 6.12 are actually vulnerable due to a logic error in the code. The present code will "only disallow" the reading of files from the white-list rather than only allowing the reading of the white-list (in effect turning it into a black-list). As such, any metrics for Path Manipulation that use false positive measurements would all be incorrect.

Program	File Name	ELOC/TLOC	Intended Vulnerable	Not Exploitable
1556	fmt1-ok.c	4/47	0	☑
1560	fmt3-ok.c	5/53	0	☑
1562	fmt5-ok.c	5/50	0	☑
1832	fmt_string_local_container_good.c	4/32	0	☑
1834	fmt_string_local_control_flow_good.c	5/36	0	☑

Table 6.13: Potential Format String issues in Dataset 3. No false positives incorrectly exploitable [0/5].

Finally, all test cases for Format String issues correctly do not contain exploitable weaknesses (Table 6.13). As such, metrics for this category that use false positive measurements would be accurate.

Discussion: Manual exploits that can be used to verify the presence of the vulnerabilities, exposed by the weaknesses, are detailed in Appendix A.

Experimental results show that while all of the programs in Test Suite 45 were in fact exploitable as intended, 13 of the programs in Test Suite 46 remained vulnerable to those weaknesses they were specifically designed to not contain (see Table 6.14).

Category	Source Code Weakness	CWE ID	TS46 Programs (SARD Test Case ID)
Input Validation	OS Command Injection	78	2136 2137 2138 2139
	Basic XSS	80	1920 1924
	Resource Injection	99	1896 1898 1900 1902
Range Errors	Heap Overflow	122	1848
Time and State	Time-of-check Time-of-use Race Condition	367	1892 1894

Table 6.14: Weaknesses found in *TS46* that violate requirement to be void of targeted CWEs.

Four *CWE-78: OS Command Injection* test cases (2136-2139) let dangerous characters pass through a filtering function that does not handle all the different command separators which exist for the command shell. This blacklist approach prevents one type of attack, but remains oblivious to many others.

Two *CWE-80: Basic XSS* test cases (1920, 1924) misuse an API function designed to securely display HTML content. Instead of using it for its intended purpose,

the program expects the function to sanitize tainted data, so it can display them. Unfortunately, data are not sanitized and can trigger a XSS attack when displayed.

All four *CWE-99: Resource Injection* test cases (1896, 1898, 1900, 1902) implement a whitelisting function to limit access to a set of files. Regrettably, the function returns inverse results, granting access to all files but the few intentionally allowed.

Test case 1848 for *CWE-122: Heap Overflow* generates a string of random size and content. It allocates memory on the heap and fills the buffer with characters. But it adds a null terminator outside the buffer leading to a buffer overflow.

Test cases 1892 and 1894, examples of *CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*, are used instead to express CWE-367: TOCTOU Race Condition. These test cases check file permissions before opening said file, however, no file locking is implemented between the time of check and the time of file opening, leading to a race condition.

Conclusion: Resulting from the above review experiment, the union of *TS45* and *TS46* is a test suite with 148 programs. While 75 programs were intended to be vulnerable, the verified number was 88 vulnerable programs. Consequently, 60 programs were not vulnerable to their stated claims (rather than 73). As a result, metrics obtained for products being evaluated for the sake of comparison will have a high error.

Category	**Source Code Weakness**	**CWE ID**	**Vulnerable Programs TS45 (SARD Test Case ID)**	**Fixed Programs TS46 (SARD Test Case ID)**
Input Validation	OS Command Injection	78	111 1881 1883 1885 [2136] [2137] [2138] [2139]	
	Basic XSS	80	1781 1794 1919 [1920] 1921 [1924] 2198	1795 1922 2204
	SQL Injection	89	1796 1798 1800	1797 1799 1801 1930
	Resource Injection	99	1895 [1896] 1897 [1898] 1899 [1900] 1901 [1902]	
Range Errors	Format String Vulnerability	134	10 92 93 1831 1833	1556 1560 1562 1832 1834

Table 6.15: Weakness categories covered by *TS45* and *TS46* following code review and verification. Test cases in [brackets] indicate those test cases, which were originally part of *TS46*, that were found to be vulnerable.

Specifically, any metrics that involve false positive data will be strongly impacted. The following subsection describes the results of a collaboration with NIST resulting from our reporting of the above (and other) inconsistencies. Furthermore, it would appear that tracking metrics for vulnerability/weakness detection in software security

assurance products may make more sense at the category level rather than across all categories in a test suite unless normalization is also applied. When each category does not contain the same number of test cases an artificial weighting is applied to the results if metrics are computed for the entire test suite. Additionally, as we have seen above, inconsistencies in the test cases (deviating from their purpose) will lead to very skewed results. Based upon our refined expectations, the expected vulnerable and not-vulnerable programs are depicted in Table 6.15.

Experiment 1 showed how our contract-based runtime monitoring could be used to verify that suspected (intended) weaknesses were in fact exploitable vulnerabilities. Experiment 2 exposed how our contract-based runtime monitoring could be used to uncover previously unknown vulnerabilities in programs that were intended to not have weaknesses. With the above five weakness categories now verified using runtime monitoring, and inconsistencies identified to understand which test cases should be exploitable, the next section will focus on the results stemming from Experiment 2 and our replacing these test suites for the larger community.

6.3.3 Improving Vulnerability Datasets

Discovery of persisting weaknesses in *TS46* led to an extensive review, update, and replacement of both *TS45* and *TS46*. The goals during the review were to improve consistency between the test suites, accuracy of obtained results, precision of test cases, and to improve the ability of researchers to automate their testing. Furthermore, the metadata around the test cases themselves were also improved. A detailed discussion of the improvements made to test suites 45 and 46, leading to the replacement test suites, along with remaining deficiencies can be found in Appendix B.

The resulting programs form test suites 100 and 101 as replacements for test suites 45 and 46. In Table 6.16 we have removed the 149 prefix from all the test case identifiers, for readability, and list all of the test case programs for each targeted CWE. Test suite 100 (TS100) contains 96 programs with intentionally seeded vulnerabilities in odd numbered test cases. Test suite 101 (TS101) contains 96 "fixed" versions of the programs from *TS100* in even numbered programs. For example, test case 148053 is a test program seeded with an *OS Command Injection* weakness. The matching pair, test case 149054, contains the same program with the weakness fixed.

Category	**Source Code Weakness**	**CWE ID**	**Vulnerable Programs TS100 (SARD Test Case ID)**	**Fixed Programs TS101 (SARD Test Case ID)**
Input Validation	OS Command Injection	78	053 151 153 155 241	054 152 154 156 242
	Basic XSS	80	093 173 175 177 215	094 174 176 178 216
	SQL Injection	89	095 097 099 187	096 098 100 188
	Resource Injection	99	157 159 161 163	158 160 162 164
Range Errors	Stack Overflow	121	055 057 059 065 067 077 087 165 167 169 193	056 058 060 066 068 078 088 166 168 170 194
	Heap Overflow	122	069 079 081 083 123 125 201	070 080 082 084 124 126 202
	Format String Vulnerability	134	045 047 061 063 111 113 237	046 048 062 064 112 114 238
	Improper Null Termination	170	127 129 131 133 195	128 130 132 134 196
API Abuse	Heap Inspection	244	085	086
	Often Misused String Management	251	137 139 141 143 145	138 140 142 144 146
Security Features	Hard-coded Password	259	105 115 117 119 121	106 116 118 120 122
Time and State	Time-of-check Time-of-use Race Condition	367	051 101 103 231 233	052 102 104 232 234
	Unchecked Error Condition	391	185	186
Code Quality	Memory Leak	401	071 073 179 181 189	072 074 180 182 190
	Unrestricted Critical Resource Lock	412	199	200
	Double Free	415	049 075 107 109 217 229	050 076 108 110 218 230
	Use After Free	416	203 219 221 223 225 239 247	172 204 220 222 224 226 240 248
	Uninitialized Variable	457	089 191 197 207	090 192 198 208
	Unintentional Pointer Scaling	468	091 183	092 184
	Null Dereference	476	147 149 209 243 245	148 150 210 244 246
Encapsulation	Leftover Debug Code	489	135	136

Table 6.16: Weakness categories, along with CWE IDs and Test Case IDs, covered by *TS100* and *TS101* (Replace *TS45* and *TS46*). The IDs of test cases start from 149045, i.e., the ID of the test case 053 will actually be 149053 (prefix 149 is removed to conserve space).

6.4 PART II: Applying Security Metrics

Experiments such as those presented in [95], have shown that static analysis is an approach that detects possible vulnerabilities more efficiently than other approaches. Using the methods outlined in [124] and [125] we have developed experiments to associate our ideas with relevant security vulnerability content.

In contrast to Section 6.3, where manual code review and runtime monitoring were employed, we employ static analysis to evaluate the impact on measurement under the assumptions held pre/post verification and the replacement test suites.

6.4.1 Experiment 3: Static Verification of Datasets

To evaluate impact of the changes, the following subsections compare the results of scanning for weaknesses against the four test suites to evaluate the following three scenarios:

> ***Base***: Consider assumptions of the test plan valid
> ***Validated***: Based upon validated test suites
> ***Replacement***: Based upon replacement test suites

The comparison is conducted using the commercial static analysis product HP Fortify SCA v6.10, under academic license, with the default 2014.1.0.0010 rulepacks. Both the *base* and *validated* scenarios are conducted against an unmodified version of *TS45* and *TS46*. The *replacement* scenario is conducted against *TS100* and *TS101*. These scenarios compare detection measurement results from the viewpoint of a software security assurance product, HP Fortify SCA, to detect the specified weakness categories under the three different scenarios. Measurements for comparison include the number of detected *true positives*, detected *false positives*, *recall*, *precision*, and *F-Measure*.

Context and Objective of Experiment 3: Run static analysis against datasets 2 and 3, as well as their replacements, to create a baseline of suspected vulnerabilities. It is widely understood that finding and fixing defects early during the development cycle reduces overall costs. Recent numbers suggest that detecting security defects early can reduce costs by 20% or more[8]. The experiment was conducted in two phases as follows: (1) Conduct static analysis to identify potential vulnerabilities. (2) Create contracts to verify a subset of the vulnerabilities.

Population: Datasets 2 and 3 are representative of a subset of well understood security vulnerabilities. There are 75 programs intended to be vulnerable in *TS45*, each of which is to exhibit one particular weakness, spanning 21 weakness categories. Additionally, *TS46* is intended to contain 73 programs, each of which is to "NOT" exhibit a particular weakness, spanning 21 weakness categories (see Table 6.2). As discovered in Experiment 2, the intention of *TS46* was not satisfied. Finally, *TS100*

[8]https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/business/267-BSI.html

consists of 96 vulnerable programs and *TS101* 96 programs that are not vulnerable (to their stated weaknesses).

Experimental Objective and Theory: Test suites produce reliable metrics for making decisions. Commercial static analysis tools identify many potential vulnerabilities in a time efficient manner but with a high rate of false positives. The stated goal of this experiment *is to identify deficiencies measured by traditional metrics and use a commercial static analysis tool to provide a baseline of static analysis efficiency for future comparison and to identify which vulnerabilities are verifiable, with contract-based security assurance monitoring, using static analysis input.* Collected data will include the following: false positive count, false negative count, true positive count, recall, precision, F1 measure, and percentage of vulnerabilities verified with contract-based runtime monitoring.

Base Scenario

This scenario holds certain assumptions under the context of the test plan defined in NIST SP 500-270 v1.1 [122]. For *TS45* these assumptions include that each program contains an instance of its stated CWE and that the weakness is exploitable. Test suite 46 holds the assumption that for each test case, there will be a "fixed" version of a program that contained the stated CWE and that the weakness is not exploitable. For both test suites, we also assume that all of the test cases are included in the download from SARD.

Scanning *TS45* and *46*, under the assumptions of the test plan, correctly identifies 74% of the weaknesses seeded in *TS45*. As shown in the *Base* column of Table 6.17, 100% of the command injection, SQL injection, resource injection, heap overflow, format string, improper null termination, heap inspection, often misused string management, memory leak, double free, use after free, and uninitialized variable vulnerabilities are detected. The results for CWE-251 are reported, as expected, with either a stack-based or heap-based buffer overflow in addition to the misused string management since CWE-251 is dangerous because it leads to buffer overflows (true in 100% of CWE-251 test cases). Even when CWE-251 is not exploitable, it is advisable to review as it could lead to future vulnerabilities.

Source Code Weakness	CWE	Base TS45/TS46 (TP)	Base TS45/TS46 (FP)	Validated TS45/TS46 (TP)	Validated TS45/TS46 (FP)	Replacement TS100/TS101 (TP)	Replacement TS100/TS101 (FP)
OS Command Injection	78	4/4	4/4	8/8	0/0	5/5	5/5
Basic XSS	80	0/5	0/5	0/7	0/3	0/5	0/5
SQL Injection	89	3/3	4/4	3/3	4/4	4/4	4/4
Resource Injection	99	4/4	4/4	8/8	0/0	4/4	4/4
Stack Overflow	121	6/9	0/8	6/9	0/8	8/11	2/11
Heap Overflow	122	4/4	2/6	5/5	1/5	6/7	1/7
Format String Vulnerability	134	5/5	0/5	5/5	0/5	7/7	0/7
Improper Null Termination	170	5/5	2/4	5/5	2/4	5/5	3/5
Heap Inspection	244	1/1	0/0	1/1	0/0	1/1	0/1
Often Misused String Management	251	5/5	0/5	5/5	0/5	5/5	0/5
Hard-coded Password	259	2/5	0/4	2/5	0/4	4/5	0/5
TOCTOU Race Condition	367	0/3	2/2	2/3	0/2	4/5	0/5
Unchecked Error Condition	391	0/1	0/1	0/1	0/1	0/1	0/1
Memory Leak	401	2/2	0/5	2/2	0/5	4/5	0/5
Unrestricted Critical Resource Lock	412	0/1	0/1	0/1	0/1	0/1	0/1
Double Free	415	6/6	1/4	6/6	1/4	5/6	1/6
Use After Free	416	5/5	0/4	5/5	0/4	6/7	0/7
Uninitialized Variable	457	3/3	0/1	3/3	0/1	4/4	0/4
Unintentional Pointer Scaling	468	0/1	0/1	0/1	0/1	0/2	0/2
Null Dereference	476	2/4	1/4	2/4	1/4	3/5	2/5
Leftover Debug Code	489	0/1	0/1	0/1	0/1	0/1	0/1
	Total	57/77	20/73	68/88	9/62	75/96	22/96
		74%	27%	77%	15%	78%	23%

Table 6.17: Programs reporting targeted CWEs by HP SCA in *TS45* and *TS46* with test plan assumptions (Base), *TS45* and *TS46* with validated assumptions (Validated), and *TS100* and *TS101* replacements (Replacement).

Twenty test programs do not report their expected weaknesses and are treated as false-negatives. Specifically, basic XSS (CWE-80) is not detected in any of the five related test programs due to HP Fortify SCA not providing specific rule support for the CGIC APIs. Issues reported as CWE-121 are only considered true positives if the analyzer correctly identified a stack-based buffer overflow, with the exception of 1563 which uses the function *gets* which is never safe, at the correct location of the weakness resulting in a 67% detection rate. Hard-coded password, CWE-259, is detected in 40% of the possible five test programs. Upon further analysis, two of the three undetected test programs (1839 and 1841) would be extremely difficult for any static analysis approach to find correctly since the source code does not directly imply the intended business logic. The hard-coded string in the conditional could have nothing to do with a password, and could be any string comparison leading to any number of possible operations. For example, the following code from 1841 could

be logic to determine if logging is enabled:

```
1   static bool logged = false;
2   int main(int argc, char *argv[]) {
3       for (unsigned i=1;i<argc && !logged;++i)
4       {
5           if (!strcmp(argv[i], "0xDEADBEEF"))
6           {
7               logged = true;
8               printf("Logged in\n");
9           }
10      }
11      return 0;
12  }
```

It would be reasonable to only detect the above hard-coded password using custom rules, provided business logic understanding, for a given static analysis product. In fact, while the above code is not desirable in enterprise code, from a maintenance standpoint it is entirely possible to have such business logic and security assurance products should have customizable features to allow users to detect these types of issues (even if the result generates high rates of false positives).

None of the three TOCTOU (CWE-367) issues are detected, however, programs 1806 and 1808 appear to be designed to demonstrate CWE-362 rather than CWE-367. For several CWEs that only had a single test case, HP Fortify SCA did not report anything {CWE-391, CWE-412, CWE-468, and CWE-489}. Finally, only two of the four Null Dereferences were reported (CWE-476).

Several CWE categories, shown in the *Base* column of Table 6.17, suffer high rates of false positives in *TS46* (e.g. CWE-78, CWE-89, and CWE-90). The cause of the false positives is the nature of the detection mechanism. All of these weaknesses are detected by dataflow analysis, which tracks taint from a data input source to a potentially vulnerable sink where the tainted data is used. If the mechanism which removes the risk of the weakness is a custom function that modifies the tainted data, such as a white-list, then the analyzer would need to be informed via a custom rule that the particular taint being tracked is removed. Specifically, all of the SQL Injection fixes are accomplished using the function `mysql_real_escape_string` that can still be vulnerable if incorrect quotations are used. The OS Command Injection issues all use custom white listing function `check` or `purify`. Finally, Resource Injection issues also use custom white-listing function `allowed`. Since HP Fortify SCA is a "security

review" product, it is debatable whether or not these issues are false positives, or not, and should be shown. The primary question is how one ensures that the business logic of the defensive function(s) truly removes all risk? Having a manual code review and penetration test to verify the functionality could lead to custom cleanse rules which remove the taint that is being tracked by the dataflow analyzer when program logic passes through the defensive function. If a custom rule is added, and the removal of taint is truly warranted, then these "false-positives" will go away. However, adding cleanse rules to remove taint should be done with care since the removal of taint can lead to false-negatives if the defensive function is insufficient (e.g. several defensive functions in *TS46* are vulnerable as discussed in Section 6.3.3).

Validated Scenario

If we assume that the test suites satisfied all the assumptions in the previous scenario, we would be almost done; however, our verification of *TS46* required us to (in)validate those assumptions. Interesting results were found by the scanner in *TS46*, under the verified test suites, including the following:

- 11 unexpected exploitable vulnerabilities {1848, 1892, 1894, 1896, 1898, 1900, 1902, 2136, 2137, 2138, 2139}.
- 9 false positives {1615, 1797, 1799, 1801, 1830, 1855, 1930, 2012, 2195}.
- 55 true negatives.

This brings the grand total to 68 true positives and 9 false positives being reported. In order to represent these validated test cases, we treat *TS45* and *TS46* as a union (see *Validated* column in Table 6.17 that highlights above changes). In addition, while verifying suspected vulnerabilities using runtime monitoring, an additional two unexpected vulnerabilities were also detected (1920 and 1924) in CWE-80, which are counted here but not detected since they would require custom rules (for a total of 13 unexpected exploitable test cases). To summarize, for the unmodified version of *TS46*, HP Fortify correctly identifies exploitable vulnerabilities in 11 programs, of the 13 identified in Section 6.3, which are not supposed to contain weaknesses.

In particular, CWE-78 and CWE-99 results vary strongly from the test plan since the programs in *TS46* were correctly identified as having weaknesses. The existence of the weaknesses in *TS46*, results in an absolute error of 13, yielding a possible percentage error of 18%. This error in *TS46* explains the almost doubling of percentage of false positives reported, from 15% to 27%, between the test suites

in the *Base* and *Validated* columns of Table 6.17. These inaccuracies also cause the increase in true positives from 74% to 77% in the validated scenario.

Four Command Injections are reported in *TS46* that should be false positives (as discussed in Section B.1). On detailed analysis, however, these appear to be true positives as the added custom function (`purify`) that is intended to remove ";" is not sufficient to remove command injection vulnerabilities. As such these tests are still exploitable, resulting in a false positive count of [0/4] when scanned with SCA. The four Resource Injection programs discussed in Section B.1 are true positives due to a logic error in the code. The present code will "only disallow" the reading of files from the white-list rather than only allowing the reading of the white-list (in effect turning it into a black-list).

In addition, during the design of an application, an architect identifies various programming languages, frameworks, and APIs that satisfy the requirements of the project. Depending upon the architectural design and implementation decisions, a particular software security assurance product may, or may not, have the ability to detect a particular weakness. For example, the use of CGIC in the test suites made it initially appear that HP Fortify was not able to detect related weaknesses, however, it was entirely possible to find these weaknesses by providing custom dataflow rules to the product which provide the necessary semantics around the APIs use that relate to security weaknesses. As a result, no XSS issues are reported in *45* or *TS46* when using the default rule set since HP SCA does not have rule support for the CGIC API (see Validated column of Table 6.17).

Finally, all four SQL Injection test cases in *TS46* appear to avoid SQL Injection attacks through a combination of using the `mysql_real_escape_string` function to escape special characters for the character set the server is expecting, along with necessary placing of single quotes (') around the variables in the query. If single quotes were not employed these would still be vulnerable (even with the use of `mysql_real_escape_string`). For example, the following would result in all rows in 'users' table.

```
[Define query string without apostrophes around %s:]
char *fmtString = "SELECT * FROM users WHERE firstname LIKE %s";

[Executing the program with:]
# sql_select -good "1 or 1=1"
```

[If this was a CGI script with encoded parameters:]
http://192.168.20.1/cgi−bin/xss_sql_select −good.cgi?q=1%20or%201=1

While these four test cases are not vulnerable to most attacks, a better defense is the use of prepared statements with parameterized queries to also avoid wildcard attacks. For example, the good versions of these test cases can still be manipulated to return more than they intend by executing a query with a wildcard parameter such as "sql_select-good %".

Replacement Scenario and Comparison

Test suites 45 and 46 had deficiencies that required a thorough analysis, discussed in Section 6.3.3, resulting in the creation of *TS100* and *TS101* as replacements. The *Replacement* column of Table 6.17 provides an assessment of coverage, reporting 78% of all true positives and a realistic 23% for false positives. The improved accuracy is a direct result of the changes, described in appendix Section B.1, that resulted in the fixing of 13 vulnerable test cases and creation of equivalent test cases grouped in GOOD/BAD pairs across the two new test suites.

Results, however, should not be interpreted as an average across all test cases since this assumes that all test cases and weakness categories are equal. Thus, the true positive and false positive rates represented at the bottom of Replacement column of Table 6.17 should not be used as the truth for any particular situation. It would be more beneficial to improving efficacy to evaluate each weakness category independently. Basic XSS, for example, appears to have no true positives nor false positives which indicates that no coverage exists. SQL Injection, on the other hand, shows a 100% true positive rate in TS100 but also a 100% false positive reporting rate in TS101. Both of these example measures indicate that improvements could be obtained for an overall security assurance program.

	TP	**FP**	**FN**	**TP+FP**	**TP+FN**	**R**	**P**	**F1**
Base	57	20	20	77	77	.74	.74	.74
Validated	68	9	20	77	88	.77	.88	.82
Replacement	75	22	21	97	96	.78	.77	.78

Table 6.18: Metrics for targeted CWEs by HP SCA in *TS45* and *TS46* under base scenario.

The final metrics, displayed in Table 6.18, summarize the results of scanning under the three different scenarios. The base and validated scenarios show an 8% increase in the F-Measure and a 14% increase in precision which can be attributed to incorrect

classification of vulnerable programs into a not-vulnerable bucket. The metrics for the replacement scenario vary from the base with a 4% increase in both recall and F-Measure, along with a 3% increase in precision. This is not the complete story, however, since the base scenario does not contain accurate metrics due to inaccuracies in TS46 and missing equivalence GOOD/BAD pairs with TS45. We predict that these differences could vary greatly when evaluating other products since the number of *TP*, *FP*, and *FN* will also vary, resulting in recall, precision, and F-Measure deltas. Finally, the number of true positives in this evaluation could be increased, and false positives significantly reduced, by employing custom rules to improve detection and dataflow results (as discussed in Section 6.4.1). Results for the the replacement scenario SCA scans can be found in Appendix E.

6.4.2 Applying Alternate Evaluation Metrics

In order to set the stage for our next experiment, we first present how the alternative metrics introduced in Section 5.2 can be applied to the results obtained in Experiment 3 for the Replacement test suites (TS100/TS101). We will show, in order, how the arithmetic mean can be applied to a single category, how artificial scaling affects the perception of coverage, and how a weighted mean can be used to align a test suite to a particular consumer's security needs. The results of these observations are summarized in Table 6.19 where the *Default* column represents the unaltered results from Experiment 3, the *Scaled* column shows artificial scaling, and the *Weighted* column shows an example of consumer-based weighted mean.

Arithmetic mean is typically applied, per weakness category, to obtain the value of Recall (or true positive rate). For example, for *OS Command Injection* we know that there are five *TP* test cases and thus have the following for the *Default* results in *TS100*:

$$A = \frac{1}{5}\sum_{i=1}^{5} TC_i = 5/5$$

Furthermore, computing the *TP Rate* across all test cases would lead us to 78% as depicted in the *Default* column in Table 6.19. Equivalently, we find a *FP Rate* of 23%.

Artificial scaling should be avoided in any comparative analysis. Observe in Table 6.19, by comparing the values for "Percentage Mean" in the *Default* column with the

Source Code Weakness	CWE	Default TS100/TS101 (TP)	Default TS100/TS101 (FP)	Scaled TS100/TS101 (TP)	Scaled TS100/TS101 (FP)	Weighted TS100/TS101 (TP)	Weighted TS100/TS101 (FP)
OS Command Injection	78	5/5	5/5	11/11	11/11	5/5	5/5
Basic XSS	80	0/5	0/5	0/11	0/11		
SQL Injection	89	4/4	4/4	11/11	11/11		
Resource Injection	99	4/4	4/4	11/11	11/11	4/4	4/4
Stack Overflow	121	8/11	2/11	8/11	2/11	8/11	2/11
Heap Overflow	122	6/7	1/7	9.43/11	1.57/11	6/7	1/7
Format String Vulnerability	134	7/7	0/7	11/11	0/11	7/7	0/7
Improper Null Termination	170	5/5	3/5	11/11	6.6/11	5/5	3/5
Heap Inspection	244	1/1	0/1	11/11	0/11	1/1	0/1
Often Misused String Management	251	5/5	0/5	11/11	0/11	5/5	0/5
Hard-coded Password	259	4/5	0/5	8.8/11	0/11		
TOCTOU Race Condition	367	4/5	0/5	8.8/11	0/11		
Unchecked Error Condition	391	0/1	0/1	0/11	0/11		
Memory Leak	401	4/5	0/5	8.8/11	0/11	4/5	0/5
Unrestricted Critical Resource Lock	412	0/1	0/1	0/11	0/11		
Double Free	415	5/6	1/6	9.17/11	1.83/11	5/6	1/6
Use After Free	416	6/7	0/7	9.43/11	0/11	6/7	0/7
Uninitialized Variable	457	4/4	0/4	11/11	0/11		
Unintentional Pointer Scaling	468	0/2	0/2	0/11	0/11		
Null Dereference	476	3/5	2/5	6.60/11	4.40/11	3/5	2/5
Leftover Debug Code	489	0/1	0/1	0/11	0/11	0/1	0/1
	Total	75/96	22/96	157/231	49/231	68/88	9/62
Arithmetic Mean		78%	23%	68%	21%	86%	26%
Percentage Mean		68%	21%	68%	21%	82%	27%

Table 6.19: Programs reporting targeted CWEs by HP SCA, with default rulepacks, in *TS100* and *TS101* and different metrics (Default, Scaled, Weighted). The elements highlighted in grey have had a weight of '0' applied to them while all others have a weight of '1'.

"Percentage Mean" in the *Scaled* column, that the values are equivalent for "Percentage Mean". Notice how the "Aritmetic Mean" and "Percentage Mean" in the *Scaled* column have the same values as well. This is caused by artificial, or implied, scaling that occurs if an evaluation is conducted by assuming that all weakness categories (the rows) should be treated equally (as discussed in Section 5.2.2). Such an approach, and the use of Percentage Mean, should be avoided because it causes skewing of results since the denominator of each category contains different sets of heterogeneous test cases. For example, consider an evaluation that only considers the categories of OS Command Injection, Heap Injection, and Unchecked Error Condition from TS100 and TS101 (see Table 6.20). Since the test suite does not have the same number of test cases for Heap Inspection or Unchecked Error Condition, compared to OS Command Injection (nor does it span the same coding complexities), artificial scaling potentially introduces a large amount of error. Each weakness category which has only a single test case, essentially becomes a binary condition that will increase or decrease the resulting true and false positive rates by the greatest common denominator for each occurrence. If a test suite, consisting of only these three categories from TS100 and

TS101, were to be evaluated using artificial scaling rather than arithmetic mean the true positive rate would be off by 46% and the false positive rate would be off by 38%.

As a result, we recommend computing metrics on a per category basis, never use the percentage mean, and only stakeholders should apply weighting to prioritize results according to their requirements. Additionally, if multiple tools are being compared, the same weighting must be used for categories across all tool evaluations to ensure consistency.

		Default TS100/TS101		**Scaled TS100/TS101**		**Weighted TS100/TS101**	
Source Code Weakness	**CWE**	**(TP)**	**(FP)**	**(TP)**	**(FP)**	**(TP)**	**(FP)**
OS Command Injection	78	5/5	5/5	5/5	5/5	5/5	5/5
Heap Inspection	244	1/1	0/1	1/5	0/5	1/1	0/1
Unchecked Error Condition	391	0/1	0/1	0/5	0/5		
	Total	6/7	5/7	6/15	5/15	6/6	5/6
Arithmetic Mean		86%	71%	40%	33%	100%	83%
Percentage Mean		40%	33%	40%	33%	60%	50%

Table 6.20: Effects of artificial scaling and consumer-based weighting, highlighted using only two categories.

In contrast, using a consumer-provided weighting system as input to an evaluation based upon arithmetic mean can provide more relevant metrics suited to a particular stakeholder. For the sake of example, we envision a stakeholder who does not have a Web interface nor SQL database as part of their application architecture. Consequently, the weight of '0' has been applied for *Basic XSS*, *SQL Injection*, *Hard-coded Password*, *TOCTOU Race Condition*, *Unchecked Error Condition*, *Unrestricted Critical Resource Lock*, *Uninitialized Variable*, and *Unintentional Pointer Scaling*. For the remaining categories a weight of '1' has been applied to keep their default weighting. As a result, only 13 of the 21 weakness categories are considered in the *Weighted* super-column of Table 6.19. The *TP Rate* and *FP Rate* for that comparison would be 86% and 26%, respectively. We assert that a framework permitting stakeholder defined weighting would be useful for conducting evaluations. A simpler example can also be seen in Table 6.20.

Using metrics which bring focus to the areas of strength and weakness for a particular security assurance program provides evidence that can lead to future improvements. Specifically, we should not use artificial scaling and stick to arithmetic means for true positive and false positive rates per weakness category. The evidence above, in Table 6.20, would indicate that the product used for security assurance

testing may need calibration in order to address the false positive rate for OS Command Injection and the true positive rate for Unchecked Error Condition. However, if the stakeholder weighted their analysis to not include Unchecked Error Condition as being security relevant according to their security policy, they would only need to calibrate to improve OS Command Injection false positive rates. In the following experiment, we will apply these metrics in order to measure and improve practices in a security assurance program.

6.4.3 Experiment 4: Improved Security Assurance

As discussed in Section 6.4.1, when security assurance techniques indicate deficiencies in existing approaches, it is possible to take a stance of continuous improvement by extending the capabilities of existing approaches with new security content.

To evaluate the efficacy of the VDS metric, the following subsections compare the results of the baseline scan with an updated scan which includes custom rules that were written following our runtime evaluation, using CB_SAMF contracts for runtime monitoring, and manual code review exercises which led to the creation of the new test suites. The comparison is conducted using the commercial static analysis product HP Fortify SCA v6.10, under academic license, with the default 2014.1.0.0010 rulepacks and one additional custom rulepack (see Appendix F). The *baseline* scenario is conducted against *TS100* and *TS101*. The *updated* scenario is conducted with the same configuration as the *baseline* with the addition of the custom rulepack. These scenarios compare detection measurement results from the viewpoint of a software security assurance product, HP Fortify SCA, to detect the specified weakness categories under the three different scenarios. Measurements for comparison include the number of detected *true positives*, detected *false positives*, *recall*, *precision*, and *F-Measure*.

Context and Objective of Experiment 4: Earlier experiments led us to understand that test suites need to be validated before being considered for use to ensure accuracy of results. Contract-based runtime monitoring proved useful in verifying inconsistencies discovered between the specification of a test suite and its implementation. A static analysis product was useful for identifying a high percentage of the weaknesses present in the test suite, however, there were still false negatives and false

positives that led key metrics to under perform.

Focusing on the five selected categories, we recall the difference between the *Base*, *Validated*, and *Replacement* (i.e. TS100/101) versions of *TS45* and *TS46*. Table 6.17 shows the True Positive and False Positive Rates for the different scenarios. The *Base* scenario can be viewed as an application, which a team presumes all weakness are known (incorrectly) as the result of code review and the use of a static analysis product. Let us assume that all of the true positive and false positive test cases are suspected weaknesses. Our *Validated* scenario can be viewed as the application after validation has been performed against the assumptions held in *Base*, while the *Replacement* scenario is simply shown for completeness. For each scenario we consider the union of both test suites to represent the application under test.

		Base TS45/TS46		**Validated TS45/TS46**		**Replacement TS100/TS101**	
Source Code Weakness	**CWE**	**(TP)**	**(FP)**	**(TP)**	**(FP)**	**(TP)**	**(FP)**
OS Command Injection	78	4/4	4/4	8/8	0/0	5/5	5/5
Basic XSS	80	0/5	0/5	0/7	0/3	0/5	0/5
SQL Injection	89	3/3	4/4	3/3	4/4	4/4	4/4
Resource Injection	99	4/4	4/4	8/8	0/0	4/4	4/4
Format String Vulnerability	134	5/5	0/5	5/5	0/5	7/7	0/7
	Total	16/21	12/22	24/31	4/12	20/25	13/25
True/False Positive Rate		76%	55%	77%	33%	80%	52%

Table 6.21: Programs, spanning five weakness categories, reporting targeted CWEs found by HP SCA in *TS45* and *TS46* with test plan assumptions (Base), *TS45* and *TS46* with validated assumptions (Validated), and *TS100* and *TS101* replacements (Replacement).

Computing both the *VCS* and *VDS* for our *Base* and *Validated* scenarios, across the five selected categories, provides a target for security assurance improvements within the SDLC.

From the validated column of Table 6.21, we can derive the values for VCS and VDS displayed in Table 6.22. Under the base scenario, no suspected vulnerabilities had been verified, resulting in all of the numerators for *VCS* being '0'. After validation, *VCS* depicts the number of vulnerabilities that were verified as exploitable out of the population of suspected vulnerabilities. For example, only five out of 10 *Format String Vulnerabilities* were verified, while seven out of zero XSS were verified (indicating there is an issue with detection since there should be a non-zero denominator). On the other hand, *VDS* shows that the combination of static analysis and code review, and verification, were able to identify all vulnerabilities except XSS. VCS helps indicate

areas for improvement relating to reducing false positive rates while VDS helps identify areas to improve false negative rates.

		Base Metrics		Validated Metrics	
Source Code Weakness	**CWE**	**(VCS)**	**(VDS)**	**(VCS)**	**(VDS)**
OS Command Injection	78	0/8	8/8	8/8	8/8
Basic XSS	80	0/0	0/7	7/0	0/7
SQL Injection	89	0/7	3/3	3/7	3/3
Resource Injection	99	0/8	8/8	8/8	8/8
Format String Vulnerability	134	0/5	5/5	5/5	5/5

Table 6.22: Programs spanning five categories reporting targeted CWEs by HP SCA, showing VCS and VCD, in *TS45* and *TS46*.

We will run a static analysis product against Test Suites 100 and 101 to compare expanded custom rules coverage against a baseline of suspected vulnerabilities. Over time deficiencies in security content coverage can be identified by tracking *VCS* and *VDS*. The experiment was conducted in two phases as follows: (1) Conduct static analysis to identify potential vulnerabilities and combine results from our contract-verified test suites, to set baseline values for *VCS* and *VDS* across several vulnerability types. (2) Create a custom rulepack to address a subset of the deficiencies, rescan the project with the additional security content, and then compare resulting metrics. As per Section 3.5, we target the following five weakness types during this analysis to provide custom rules to address deficiencies discovered in our earlier experiments: Format String, Resource Injection/Path Manipulation, OS Command Injection, SQL Injection, and Basic XSS.

Population: *TS100* consists of 96 vulnerable programs and *TS101* of 96 programs that are not vulnerable (to their stated weaknesses), across 21 vulnerability categories.

Experimental Objective and Theory: Test suites produce reliable metrics for making decisions. Commercial static analysis tools identify many potential vulnerabilities in a time efficient manner but with a potentially high rate of false positives and potential for false negatives. *Proposed metrics, VCS and VDS, can be used to identify deficiencies and measure improvements to security assurance coverage in an SDLC.* The stated goal of this experiment is to identify and remediate deficiencies in a commercial static analysis product using the contract-based runtime verified results from the test suites in order to measurably improve the security assurance coverage

for future projects. An example of such a deficiency is Fortify's inability to detect the five XSS weaknesses, caused by the fact that the C-based API which is used in the CGI script is not a supported API of the default Fortify rulepacks. By writing custom rules for these APIs, we are able to reach 100% detectability. Collected data will include the following: false positive count, false negative count, true positive count, recall, precision, F1 measure, VCS, and VDS.

Results: Building upon the baseline results of our revised test suites and the verification of a subset of the categories (using the approach from Figure 3.1), we have identified deficiencies in the coverage provided by HPE Fortify SCA when run against *TS100* and *TS101*. Deficiencies are highlighted by the low values in the VCS and VDS columns for the Baseline Metrics in Table 6.25. The functionality of Fortify SCA combines static analysis analyzers with security content (specifying the rules used to identify weaknesses). In order to further improve security assurance, we need to consider how to refine either the analyzers themselves or the security content. Since we do not have access to the source code of the analyzers, we will focus on creating a custom rulepack to specify additional rules to improve the results. Specifically, we need to consider new rules to increase true positives to rectify the low VDS for Basic XSS (thus reducing false negatives), and reduce false positives to improve the low VCS for all categories except Format String Vulnerability.

In total we created 12 custom rules to accomplish the above goals (available in Appendix F). The complete results, comparing the baseline scan of *TS100* and *TS101* using the default rulepacks against an *updated* scan that used custom rules can be found in Table 6.23. Consequently, we were able to remove all but one of the false positives and improve detection to 100% for the five targeted categories. To remove the remaining false positive would require access to the SCA analyzer and cannot be addressed by custom rules.

Comparing against the initial metrics from earlier experiments, we now detect 83% (from 78%) of all vulnerabilities in *TS100* and report 10% (from 23%) of false positives in *TS101*. Table 6.24 compares the higher level metrics across the evolution of the test suites. For *TS100* and *TS101*, recall has improved from 78% to 83%, precision from 77% to 89%, and F1 from 78% to 86%.

Alternatively, if we use the theory from Section 5.2 and apply weighting, we could

Source Code Weakness	CWE	Baseline TS100/TS101 (TP)	Baseline TS100/TS101 (FP)	Updated TS100/TS101 (TP)	Updated TS100/TS101 (FP)
OS Command Injection	78	5/5	5/5	5/5	1/5
Basic XSS	80	0/5	0/5	5/5	0/5
SQL Injection	89	4/4	4/4	4/4	0/4
Resource Injection	99	4/4	4/4	4/4	0/4
Stack Overflow	121	8/11	2/11	8/11	2/11
Heap Overflow	122	6/7	1/7	6/7	1/7
Format String Vulnerability	134	7/7	0/7	7/7	0/7
Improper Null Termination	170	5/5	3/5	5/5	3/5
Heap Inspection	244	1/1	0/1	1/1	0/1
Often Misused String Management	251	5/5	0/5	5/5	0/5
Hard-coded Password	259	4/5	0/5	4/5	0/5
TOCTOU Race Condition	367	4/5	0/5	4/5	0/5
Unchecked Error Condition	391	0/1	0/1	0/1	0/1
Memory Leak	401	4/5	0/5	4/5	0/5
Unrestricted Critical Resource Lock	412	0/1	0/1	0/1	0/1
Double Free	415	5/6	1/6	5/6	1/6
Use After Free	416	6/7	0/7	6/7	0/7
Uninitialized Variable	457	4/4	0/4	4/4	0/4
Unintentional Pointer Scaling	468	0/2	0/2	0/2	0/2
Null Dereference	476	3/5	2/5	3/5	2/5
Leftover Debug Code	489	0/1	0/1	0/1	0/1
	Total	75/96	22/96	80/96	10/96
Arithmetic Mean		78%	23%	83%	10%

Table 6.23: Programs reporting targeted CWEs by HP SCA, with default rulepacks (Baseline) and custom rulepack (Updated), in *TS100* and *TS101*. Changes caused by custom rules highlighted in gray.

assume our stakeholder is only interested in five of the 21 vulnerability types (Format String, Resource Injection/Path Manipulation, OS Command Injection, SQL Injection, and Basic XSS). Computing the metrics leads to strong improvements in all metrics as shown in Tables 6.25 and 6.26.

Finally, comparing the baseline Vulnerability Confidence Score (*VCS*) and Vulnerability Detection Score (*VDS*) (in Table 6.25) we can see that the baseline configuration for Fortify SCA is very capable of finding all of the categories in the defined subset of the test suites except for XSS.

While the VCS shows high confidence that the suspected vulnerabilities are exploitable (due to runtime monitoring and exercising), the VDS is presently 0% for XSS and is less than 100% for three other categories indicating higher false positive rates (OS Command Injection, SQL Injection, and Resource Injection). From these observations we are able to prioritize and improve these deficiencies by adding the custom rules that resulted in us now finding 100% of the XSS weaknesses (VCS

	TP	FP	FN	TP+FP	TP+FN	R	P	F1
Base	57	20	20	77	77	.74	.74	.74
Validated	68	9	20	77	88	.77	.88	.82
Replacement	75	22	21	97	96	.78	.77	.78
Updated	80	10	16	90	96	.83	.89	.86

Table 6.24: Metrics for targeted CWEs by HP SCA in *TS100* and *TS101* comparing results before (Replacement) and after custom rules (Updated). Initial results from Experiment 6.4.1 are also included for comparison.

		Baseline TS100/TS101		Updated TS100/TS101		Baseline Metrics		Updated Metrics	
Source Code Weakness	**CWE**	**(TP)**	**(FP)**	**(TP)**	**(FP)**	**(VCS)**	**(VDS)**	**(VCS)**	**(VDS)**
OS Command Injection	78	5/5	5/5	5/5	1/5	5/10	5/5	5/6	5/5
Basic XSS	80	0/5	0/5	5/5	0/5	0/0	0/5	5/5	5/5
SQL Injection	89	4/4	4/4	4/4	0/4	4/8	4/4	4/4	4/4
Resource Injection	99	4/4	4/4	4/4	0/4	4/8	4/4	4/4	4/4
Format String Vulnerability	134	7/7	0/7	7/7	0/7	7/7	7/7	7/7	7/7
	Total	20/25	13/25	25/25	1/25				
Arithmetic Mean		80%	52%	100%	4%				

Table 6.25: Programs spanning five categories reporting targeted CWEs by HP SCA, with default rulepacks (Baseline) and custom rulepack (Updated), in *TS100* and *TS101*. Changes caused by custom rules highlighted in gray.

identified deficiency), resulting in a 100% VDS value once the new rules are integrated. Additionally, custom rules were also written to reduce the false positives, resulting in only OS Command Injection with less than 100% (5/6) in the *Updated* set.

Summary: Using the metrics of *VCS* and *VDS*, in addition to the other metrics presented in this chapter, we further our ability to mature security assurance efforts over time. Using different approaches to identify potential vulnerabilities, the above experiment shows the feasibility of creating base assumptions where no vulnerabilities have been verified for a given vulnerability category. With no vulnerabilities verified, there is no confidence yet in the VCS. Once verification of suspected vulnerabilities is conducted, a subset (potentially complete) of the suspected vulnerabilities will be verified leading to a percentage greater, or equal, to zero percent. Ideally, we would want a *VCS* score of 100% for every category, however, as we can see from the initial verification of *TS45* and *TS46* the *VCS* indicates issues in one out of the five vulnerability categories being detected by the static analyzers as having false positives (SQL Injection). When we consider *TS100* and *TS101*, however, three of the five categories suffered from false-postive detection (OS Command Injection, SQL Injection, and Resource Injection). Ultimately, it would have been impossible to

	TP	FP	FN	TP+FP	TP+FN	R	P	F1
Replacement	20	13	5	33	25	.80	.61	.69
Updated	25	1	0	26	25	1.00	.96	.98

Table 6.26: Metrics for targeted CWEs by HP SCA in *TS100* and *TS101* comparing results before (Replacement) and after custom rules (Updated) for only the stated five categories in TS100 and TS101.

detect the false-positives in *TS46* since the test cases themselves were erroneously still vulnerable and examples of false positives for potential detection simply didn't exist.

The base assumption for each category is that no vulnerabilities have been detected even though they likely exist. When a zero-day vulnerability (a verified and previously unknown vulnerability) is found and reported for a particular software artifact we know that the vulnerability exists even though we don't yet detect it. Over time the vulnerability knowledge for each software artifact evolves and *VDS* becomes non-zero. In relation to the union of *TS100* and *TS101* in the experiment above, we knew from our previous work that the number of exploitable vulnerabilities in each category was fixed. In practice, however, the number of zero-day vulnerabilities in a program is obviously unknown. Thus, there will always be the potential for *VDS* and *VCS* to be misleading when considering previously unknown vulnerabilities.

Ultimately, we were able to use these metrics, along with the previous metrics, to initially track inconsistencies in test suites 45 and 46 (e.g. eight *OS Command Injection* vulnerabilities instead of the designed four, seven XSS where there should have been five, and eight resource injections instead of four). The replacement test suites were also still erroneous and we were able to track false-negative deficiencies for XSS, as well as false positive deficiencies in three categories. After adding custom rules, rules, to improve true positives and reduce false positives, the static scans were able to improve VDS and VCS to 100% for all vulnerability categories except for the one remaining false positive in *OS Command Injection* which led to a VDS of 100% (5/5) and a VCS of 83% (5/6).

These experiments show that for a specific tool, VCS and VDS can be used to increase confidence, per vulnerability category, in overall coverage and precision. Over time, these metrics can be applied across various detection mechanisms to improve overall security assurance when combined with test suites which have been either formally or empirically verified.

6.5 Experimental Summary

In order to evaluate the ability of CB_SAMF to verify the reachability and exploitability of suspected vulnerabilities for an improved software security assurance process, we required a dataset. This chapter identified Test Suites 45 and 46 from NIST as candidates for use in our experiments. The above experiments empirically evaluate the effectiveness of CB_SAMF at verifying specific suspected vulnerabilities that have been identified using approaches such as code review or static analysis. Our aim in these experiments is to understand the capabilities and limitations of static analysis tools for identifying vulnerabilities and evaluate our proposed CB_SAMF for verifying detected vulnerabilities. Lastly, we also want to understand how these capabilities and limitations can be measured and evaluated and propose a series of metrics to assist in understanding the resulting data.

The first experiment involved the creation of runtime probes to verify the exploitability of "known vulnerabilities" in TS45 for *OS Command Injection*, *Basic XSS*, *SQL Injection*, *Resource Injection*, and *Format String Vulnerability*. All 21 programs, which were evaluated, were vulnerable as expected and resulted in 100% TP verification of the targeted weaknesses.

Our second experiment was intended to validate our dataset using manual code review and verification to ensure the integrity of our results. While the test cases in TS45 were vulnerable as intended, the results showed that 13 of the 73 programs in TS46 that were designed to be "not vulnerable" were in fact still exploitable for the weaknesses they were not supposed to contain. Ten of the 13 inconsistencies were in the five vulnerability categories we were targeting related to input validation. Considering our population of 148 test programs spanning TS45 and TS46 had 88 vulnerable programs rather than 75, it was determined that a thorough evaluation of the design and implementation was in order to ensure the integrity of future experiments. Specifically, any measurements involving False Positive data will have high error.

Section 6.3.3 detailed our extensive review of the test suites, which we selected for Datasets 2 and 3, in order to improve the consistency, accuracy, precision, and automation for empirical evaluation of security assurance tools. The exercise resulted in a collaboration with the National Institute of Standards and Technology that

included the fixing of 13 incorrect test cases spanning five of the 21 vulnerability categories and the removal of 112 extraneous weaknesses. As part of an alignment of good/bad test case pairs we created 27 new test cases (along with 2 new test case pairs for alignment with targeted CWEs). To improve automated testing, we renamed files to comply with a naming convention and inserted comments in code to indicate the location of flaws. Our addition of new metadata to the SARD website allows researchers to track associations and deprecations of test cases. For each of our test cases listed in SARD, we corrected numerous syntactical and library dependency issues, included the compilation instructions for each test case to ensure consistency between researcher results, updated attribution of contributors, corrected CWE categorization of test cases, and deprecated all of the old test cases. The replacement test suites, 100 and 101, each contain exactly 96 test programs.

Following our evaluation of test suites 45 and 46, which led to the creation of test suites 100 and 101, the third experiment was conducted to evaluate how incorrect datasets impact security assurance approach evaluations. We created three logical datasets for the experiment based upon the base TS45 and TS46 with the assumptions held in their specifications, validated TS45 and TS46 (which involved re-categorizing a subset of non-vulnerable programs as vulnerable following Section 6.3.3 evaluation), and the replacement test suites 100 and 101. Using HPE Fortify SCA, under academic license, we demonstrated how measurements such as true positives, false positives, recall, precision, and F-Measure were impacted by invalid assumptions. The resulting values for the preceding metrics were impacted anywhere from 3% to 14% for the single static code analysis product due to the inaccuracies of the test suites, however, we would expect even wider variations when comparing across different products.

Evaluating the results of a static code analysis product raised other concerns with how metrics could provide misleading inferences due to the inherent design and implementation of the test suites. Section 5.2 discussed the confounding factors, which can lead to artificial scaling and weighting, and introduce a consumer weighting mechanisms that can allow stakeholders to prioritize which vulnerability types they are interested in so that metrics are specific to their needs. Ultimately, however, test suites need to have sufficient coverage for coding constructs, APIs, and vulnerabilities. Furthermore, we introduce Vulnerability Confidence Score (VCS) and Vulnerability Detectability Score (VDS) as metrics that can assist in measuring effectiveness of

software security assurance processes integrated into SDLCs.

Experiment 4 uses the weighted mean to perform an evaluation against only our five selected vulnerability categories and shows how VDS and VCS can be employed to remediate deficiencies in security assurance approaches by employing both static code analysis and runtime monitoring. VCS identified high false positive rate deficiencies in three of the five vulnerability categories while VDS indicated a complete detection deficiency for Basic XSS. Runtime verification allowed us to have confidence that specific test programs were in fact vulnerable providing validation of resulting measures. Following the identification of the deficiencies we created 12 custom rules for the static code analysis product that resulted 100% VCS for each category, 100% VDS for each category except OS Command Injection (83%). Additionally, perfect scores for all earlier metrics were obtained except for one FP in OS Command Injection, which led to an overall precision score of 96% and an F1 score of 98%.

In conclusion, contract-based runtime monitoring can be used to verify the exploitability of suspected vulnerabilities. Measuring the efficacy of vulnerability detection and verification, however, requires reliable datasets and meaningful metrics. We have shown that not only can static code analysis and runtime monitoring be combined to verify suspected weaknesses as being vulnerable, but also that they can compliment each other in order to improve security assurance effectiveness.

Edsger W. Dijkstra in 1970 wrote in "Notes On Structured Programming, corollary at the end of Section 3, On The Reliability of Mechanisms" the following:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

While these experiments show mechanisms to identify weakness in source code and verify them as exploitable vulnerabilities at runtime, it does not guarantee the absence of weaknesses nor the overall security of the artifact under study.

Chapter 7

Conclusions

The shortcut to improved security [is] universal, repeatable monitoring, ... The Army is now trying Harris STAT. The big difference is that NASA picked the most critical vulnerabilities rather than looking at all 2,000. The latter always leads to overload and lack of action. NASA's approach works.
-Alan Paller

Presently the cyber software security market is projected to reach $80.02 billion by 2017[1]. Cybercrime costs the UK an estimated £27 bn per year according to the UK Office of Cyber Security and Information Assurance[2]. A primary root cause of this market is the exploitation of vulnerabilities in both software and hardware systems. A large portion of the software security market is targeting the defense of systems using reactive measures that are aimed at treating symptoms. Antivirus, firewalls, and intrusion detection systems are examples of mechanisms that attempt to identify potential attacks using reactive measures. The long term goal, however, should not be detecting attacks but rather the engineering of systems that are resilient to attacks by eliminating potential vulnerabilities.

It is widely understood that finding and fixing defects early during the development cycle reduces overall costs [126, 127, 128]. Recent numbers suggest that detecting

[1] http://www.prweb.com/releases/cyber_security/application_content_data/prweb8262390.htm
[2] http://www.baesystems.com.au/Security/NationalSecurity/Uploads/Resource/THE_COST_OF_CYBER_CRIME_SUMMARY_FINAL_14_February_2011.pdf

security defects early can reduce costs by 20% or more[3]. Furthermore, the security industry has seen a history of design, implementation, and configuration related vulnerabilities over the years. While the industry is now putting standards, specifications, recommendations and the like into practice, we are still lacking tools and methodologies to assist in the identification and validation of vulnerabilities. Some techniques, such as static analysis, are already providing ways of identifying potential vulnerabilities during the early phases of the SDLC; however, these approaches can lead to a high rate of false-positives consuming resources. Our enhanced version of contracts provides a novel way to propagate requirements-based security assertions through the SDLC.

Our form of contract extends the work done by Barringer *et al* [44, 43] and Meyer [57] by adding mechanisms for security vulnerability monitoring using contracts. CB_SAMF has the potential to help reduce vulnerabilities in multi-layered systems by not only providing a way to detect a contract violation, but also provides reactive measures and a means for collecting forensic data. Our model is capable of spanning multiple software architectural layers through the use of breakpoint-based contracts instrumented through kernel-level infrastructure based upon Kprobes. We have shown that it is feasible to implement a monitoring system based on probes that is able to capture the exploitation of vulnerabilities even at the lowest levels of the runtime architecture (the kernel). As of the 3.5 Linux kernel the Uprobes patch has been integrated into the mainline Linux kernel. This support, along with the Kprobes patch that went into 2.6.9 provides a great deal of diversity to the Linux kernel.

We have shown that it is possible to implement a monitoring framework that captures assertion violations even at the lowest layers of a runtime architecture. The process of translating a contract into probes dynamically using a tool should, at a minimum, require the symbol/address of the breakpoint (or the source code), and the vulnerability type and code due to patterns in the weakness types. These properties will be derived from the contract and source code. Automation of many of these steps should be possible. For example, our recent experiments have shown that a static analysis tool could be used to provide the necessary information for the identification properties needed by the contract. Then the security tester can choose the other attributes of the contract before assertion probes are generated. Once contracts are

[3]https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/business/267-BSI.html

defined the probes can be dynamically generated by extending the SystemTap scripting language and evaluation could be achieved through the employment of metrics using another probing framework.

Including CB_SAMF in an integrated SDLC for verifying suspected weaknesses allows stakeholders to address not only the highest priority issues, but also those that can be confirmed as reachable. Our first experiment showed that runtime monitoring could be used to verify the reachability of vulnerabilities, spanning five categories, in 21 programs. Our second experiment investigated the validity of the test suite we selected for evaluation, leading to discovered deficiencies in the implementation. These deficiencies were addressed through a collaborative project with NIST, which spanned roughly a year, and led to the release of a pair of new test suites that replaced TS45 and TS46 in the Security Assurance Reference Dataset hosted at NIST. The primary deficiencies included 13 test cases containing exploitable weaknesses that were explicitly not intended to be vulnerable, 112 extraneous weaknesses spanning 82 test cases, and 60 incorrect good/bad test case pairings. These test suites now provide a much more accurate dataset for security research to be conducted against and provide insight to future researchers looking to create new datasets.

With the new datasets, we were then able to use our process for identifying potential weaknesses using static analysis against three versions of our selected dataset (base, validate, replacement) in order to evaluate how metrics are impacted by invalid assumptions (3% to 14% for one static analysis product). Furthermore, evaluation of the metrics within the context of the dataset led to the definition of two additional metrics that would allow us to measure the vulnerability detection score and vulnerability confidence score of a security assurance approach. We also showed how stakeholders could customize a test suite to their specific requirements using a weighting approach for each category. The last experiment showed how using the new metrics we could not only improve the detection and verification of suspected weaknesses by combining static analysis and runtime monitoring, we also showed how the security assurance process could be improved over time by combining the approaches and measuring deficiencies to identify areas for improvement.

Software Fault Injection (SFI) remains an important part of our approach that we have not addressed. Bishop, Ghosh, and Whittaker identify SFI as an important

approach for injecting faults into systems and environments but also note that they need to be combined with other approaches to cover all security-related flaws [15]. Finally, MITRE does not consider runtime monitoring approaches for detection or mitigation of CWEs. MITRE could also consider amending the CWE entries to contain runtime monitoring mechanisms, where appropriate, for both Detection Methods and Potential Mitigations.

Currently we are working on mechanisms to assist in the automation of contract creation using output results from static analysis, and code review. Such an automation would further our work by allowing continuous improvement to software detection and verification leading to improved efficiency for the industry and field. Several other challenges exist in this space including the automated creation of test suites, or datasets, which provide sufficient coverage of coding constructs, vulnerability categories, and API usage for languages that would provide the basis for verification and stakeholder evaluation.

While we have accomplished many of the goals set of for this research, the area of software security assurance is quickly evolving and there is a tremendous amount of research yet to be addressed in this area.

Bibliography

[1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[2] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft Press, 2006.

[3] M. Bishop, *Computer Security: Art and Science*. Boston, MA, USA: Addison-Wesley, 2003.

[4] M. Howard, "Building more secure software with improved development processes," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 63–65, 2004.

[5] G. Larsen, E. Fong, D. A. Wheeler, and R. S. Moorthy, "State-of-the-art resources (soar) for software vulnerability detection, test, and evaluation," tech. rep., DTIC Document, 2014.

[6] D. Geer, "For good measure: The undiscovered," *;login:*, vol. 40, April 2015.

[7] B. Schneier, "Should U.S. hackers fix cybersecurity holes or exploit them?." `http://www.theatlantic.com/technology/archive/2014/05/should-hackers-fix-cybersecurity-holes-or-exploit-them/371197/`.

[8] D. Serpanos and J. Henkel, "Dependability and security will change embedded computing," *Computer*, vol. 41, no. 1, pp. 103–105, 2008.

[9] J. M. Voas, G. McGraw, A. Ghosh, F. Charron, and M. Schatz, "Quantifying minimum-time-to-intrusion based on dynamic software safety assessment," tech. rep., Defense Technical Information Center OAI-PMH Repository [http://stinet.dtic.mil/oai/oai] (United States), 1998.

[10] P. E. Black, L. Badger, B. Guttman, and E. Fong, "NIST IA 8151. Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2016.

[11] C. Ko, M. Ruschitzka, and K. N. Levitt, "Execution monitoring of security-critical programs in distributed systems: a specification-based approach," in *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 175–187, IEEE Computer Society, 1997.

[12] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Proceedings of the 10th Annual Computer Security Applications Conference*, (Orlando, FL), pp. 134–144, IEEE Computer Society Press, 1994.

[13] A. B. Somayaji, *Operating system stability and security through process homeostasis*. PhD thesis, The University of New Mexico, 2002. Chairperson-Stephanie Forrest.

[14] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 318–329, ACM Press, 2004.

[15] J. Reynolds, M. Bishop, A. Ghosh, and J. Whittaker, "How useful is software fault injection for evaluating the security of cots products?," in *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, pp. 339–340, Dec 2001.

[16] M. Payer, *HexPADS: A Platform to Detect "Stealth" Attacks*, pp. 138–154. Cham: Springer International Publishing, 2016.

[17] M. Jouad, S. Diouani, H. Houmani, and A. Zaki, "Security challenges in intrusion detection," in *2015 International Conference on Cloud Technologies and Applications (CloudTech)*, pp. 1–11, June 2015.

[18] A. Sadeghian, M. Zamani, and S. Ibrahim, "Sql injection is still alive: A study on sql injection signature evasion techniques," in *2013 International Conference on Informatics and Creative Multimedia*, pp. 265–268, Sept 2013.

[19] D. K. Peters and D. L. Parnas, "Requirements-based monitors for real-time systems," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 77–85, 2000.

[20] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009.

[21] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, and P. Grnbacher, "A comparison framework for runtime monitoring approaches," *Journal of Systems and Software*, vol. 125, pp. 309 – 321, 2017.

[22] G. Spanoudakis and K. Mahbub, "Requirements monitoring for service-based systems: towards a framework based on event calculus," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 379–384, Sept 2004.

[23] I. Aktug and K. Naliuka, "Conspec a formal language for policy specification," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 45 – 58, 2008. Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007).

[24] H. Gunadi and A. Tiu, *Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System*, pp. 296–311. Cham: Springer International Publishing, 2014.

[25] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. v. d. Aalst, "Monitoring business constraints with the event calculus," *ACM Trans. Intell. Syst. Technol.*, vol. 5, pp. 17:1–17:30, Jan. 2014.

[26] A. M. Hoole and I. Traore, "Contract-based security monitors for service oriented software architecture," in *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, APSCC '08, (Washington, DC, USA), pp. 1239–1245, IEEE Computer Society, 2008.

[27] A. M. Hoole, I. Simplot-Ryl, and I. Traore, "Integrating contract-based security monitors in the software development life cycle," in *2nd Workshop on Formal Languages and Analysis of Contract-Oriented Software*, FLACOS 2008, (Malta), pp. 25–30, Nov 2008.

[28] A. M. Hoole, I. Traore, and I. Simplot-Ryl, "Application of contract-based security assertion monitoring framework for telecommunications software engineering," *Math. Comput. Model.*, vol. 53, pp. 522–537, Feb. 2011.

[29] A. M. Hoole, I. Traore, A. Delaitre, and C. de Oliveira, "Improving vulnerability detection measurement: [test suites and software security assurance]," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, (New York, NY, USA), pp. 27:1–27:10, ACM, 2016.

[30] A. M. Hoole and I. Traore, "Combining static analysis and runtime monitoring for verifiable security assurance metrics," *IEEE Transactions on Software Engineering*, Submitted for Publication.

[31] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 702–705, Nov 2013.

[32] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu, *MONPOLY: Monitoring Usage-Control Policies*, pp. 360–364. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[33] M. Pohlack, B. Döbel, and A. Lackorzyński, "Towards runtime monitoring in real-time systems," in *Eighth Real-Time Linux Workshop*, pp. 173–184, 2006.

[34] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[35] S. N. Chari and P.-C. Cheng, "Bluebox: A policy-driven, host-based intrusion detection system," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 2, pp. 173–200, 2003.

[36] J. Zimmerman, L. Mé, and C. Bidan, "Experimenting with a policy-based hids based on an information flow control model," in *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, (Washington, DC, USA), p. 364, IEEE Computer Society, 2003.

[37] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *15th USENIX Security Symposium*, pp. 289–304, August 2006.

[38] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, (Washington, DC, USA), pp. 48–62, IEEE Computer Society, 2006.

[39] I. Aktug, M. Dam, and D. Gurov, *Provably Correct Runtime Monitoring*, pp. 262–277. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[40] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, *Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors*, pp. 68–84. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[41] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, pp. 249–289, Jun 2012.

[42] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to aspectj," *SIGPLAN Not.*, vol. 40, pp. 345–364, Oct. 2005.

[43] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Program monitoring with ltl in eagle," *ipdps*, vol. 17, p. 264b, 2004.

[44] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," 2003.

[45] H. Barringer, A. Groce, K. Havelund, and M. H. Smith, "Formal analysis of log files," *JACIC*, vol. 7, no. 11, pp. 365–390, 2010.

[46] H. Barringer and K. Havelund, "Tracecontract: A scala DSL for trace analysis," in *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pp. 57–72, 2011.

[47] L. Wang, E. Wong, and D. Xu, "A threat model driven approach for security testing," in *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, (Washington, DC, USA), p. 10, IEEE Computer Society, 2007.

[48] Y. Falcone, S. Currea, and M. Jaber, "Runtime verification and enforcement for android applications with rv-droid," in *Runtime Verification* (S. Qadeer and

S. Tasiran, eds.), vol. 7687 of *Lecture Notes in Computer Science*, pp. 88–95, Springer Berlin Heidelberg, 2013.

[49] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.

[50] W. G. J. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter sql-injection attacks," in *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, (New York, NY, USA), pp. 1–7, ACM, 2005.

[51] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, Dec 2004.

[52] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (New York, NY, USA), pp. 411–420, ACM, 1999.

[53] L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, and W. M. van der Aalst, "Compliance monitoring in business processes," *Inf. Syst.*, vol. 54, pp. 209–234, Dec. 2015.

[54] M. T. Khan, D. Serpanos, and H. Shrobe, "A rigorous and efficient run-time security monitor for real-time critical embedded system applications," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 100–105, Dec 2016.

[55] M. B. Dwyer, R. Purandare, and S. Person, *Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?*, pp. 36–50. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[56] E. Bush, "A gold standard for assessing the coverage of static analyzers," in *2013 IEEE International Conference on Technologies for Homeland Security (HST)*, pp. 710–715, Nov 2013.

[57] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[58] J. C. M. Jr., "Programming by contract," *Computer*, vol. 29, no. 3, pp. 109–111, 1996.

[59] L. Lamport, "A simple approach to specifying concurrent systems," *Commun. ACM*, vol. 32, no. 1, pp. 32–45, 1989.

[60] A. Février, E. Najm, and J.-B. Stefani, "Contracts for odp," in *ARTS '97: Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, (London, UK), pp. 216–232, Springer-Verlag, 1997.

[61] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, "Monitoring metric first-order temporal properties," *J. ACM*, vol. 62, pp. 15:1–15:45, May 2015.

[62] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 21–28, Sept 2012.

[63] M. Abadi and L. Lamport, "Composing specifications," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 1, pp. 73–132, 1993.

[64] P. W. Oman and S. L. Pfleeger, eds., *Applying Software Metrics*. New York, NY, USA: IEEE Standards Office, 1997.

[65] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

[66] G. O'Regan, *Mathematical Approaches to Software Quality*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[67] R. Khan, K. Mustafa, and S. Ahson, *Software quality: Concepts and practices*. Alpha Science, 2006.

[68] S. Payne, *A Guide to Security Metrics, SANS Institute*, 2006 (accessed 2016). `https://www.sans.org/reading-room/whitepapers/auditing/guide-security-metrics-55`.

[69] B. Heinzle and S. Furnell, *Assessing the Feasibility of Security Metrics*, pp. 149–160. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[70] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, 1995.

[71] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, (New York, NY, USA), pp. 315–317, ACM, 2008.

[72] C. D. De Oliveira, P. E. Black, and E. Fong, "NIST IR 8165. Impact of code complexity on software analysis," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2017.

[73] S. B. Lipner, "Security assurance," *Commun. ACM*, vol. 58, pp. 24–26, Oct. 2015.

[74] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, (New York, NY, USA), pp. 37–44, ACM, 2014.

[75] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Comput. Surv.*, vol. 49, pp. 62:1–62:35, Dec. 2016.

[76] J. Bird, *Using Metrics to Manage Your Application Security Program, SANS Institute*, 2016 (accessed 2016). `https://www.sans.org/reading-room/whitepapers/analyst/metrics-manage-application-security-program-36822`.

[77] P. E. Black and E. Fong, "NIST SP 500-320. Report of the workshop on software measures and metrics to reduce security vulnerabilities (swmm-rsv)," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2016.

[78] V. Verendel, "Quantified security is a weak hypothesis: A critical survey of results and assumptions," in *Proceedings of the 2009 Workshop on New Security Paradigms Workshop*, NSPW '09, (New York, NY, USA), pp. 37–50, ACM, 2009.

[79] B. Chess and J. West, *Secure programming with static analysis*. Addison-Wesley Professional, first ed., 2007.

[80] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, "Effect of static analysis tools on software security: Preliminary investigation," in *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, (New York, NY, USA), pp. 1–5, ACM, 2007.

[81] P. E. Black, "Counting bugs is harder than you think," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pp. 1–9, Sept 2011.

[82] P. Anderson, "Improving software reliability and security with automated analysis," in *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pp. 1–6, Nov 2008.

[83] P. Anderson, "Measuring the value of static-analysis tool deployments," *IEEE Security Privacy*, vol. 10, pp. 40–47, May 2012.

[84] A. Delaitre, V. Okun, and E. Fong, "Of massive static analysis data," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pp. 163–167, June 2013.

[85] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Inf. Softw. Technol.*, vol. 68, pp. 18–33, Dec. 2015.

[86] V. Okun, A. Delaitre, and P. E. Black, "NIST SP 500-297. Report on the static analysis tool exposition (sate) iv," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2013.

[87] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against {SAMATE} tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462 – 1476, 2013.

[88] X. Yang and M. Zulkernine, "Security monitoring of components using aspects and contracts in wrappers," in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pp. 566–575, July 2011.

[89] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *Services Computing, IEEE Transactions on*, vol. 8, pp. 269–283, March 2015.

[90] M. Jimenez, M. Papadakis, and Y. L. Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pp. 105–112, Dec 2016.

[91] P. E. Black, E. Fong, V. Okun, and R. Gaucher, "NIST SP 500-269 v1.0. Software assurance tools: Web application security scanner functional specification version 1.0," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2008.

[92] P. Ferrara, E. Burato, and F. Spoto, "Security analysis of the owasp benchmark with julia," in *ITASEC*, 2017.

[93] H. H. AlBreiki and Q. H. Mahmoud, "Evaluation of static analysis tools for software security," in *2014 10th International Conference on Innovations in Information Technology (IIT)*, pp. 93–98, Nov 2014.

[94] G. Chatzieleftheriou and P. Katsaros, "Test-driving static analysis tools in search of c code vulnerabilities," in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pp. 96–103, July 2011.

[95] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pp. 451–460, Nov 2013.

[96] G. Peterson and J. Steven, "Defining misuse within the development process," *IEEE Security and Privacy*, vol. 04, no. 6, pp. 81–84, 2006.

[97] P. Hope, G. McGraw, and A. I. Anton, "Misuse and abuse cases: Getting past the positive," *IEEE Security and Privacy*, vol. 02, no. 3, pp. 90–92, 2004.

[98] O. E. Dictionary, *"issue, n."*. Oxford University Press, 2016.

[99] O. E. Dictionary, *"defect, n."*. Oxford University Press, 2016.

[100] O. E. Dictionary, *"weakness, n."*. Oxford University Press, 2016.

[101] O. E. Dictionary, *"weak, adj. and n."*. Oxford University Press, 2016.

[102] O. E. Dictionary, *"vulnerability, n."*. Oxford University Press, 2016.

[103] O. E. Dictionary, *"vulnerable, adj."*. Oxford University Press, 2016.

[104] O. E. Dictionary, *"exploit, n."*. Oxford University Press, 2016.

[105] O. E. Dictionary, *"attack, n."*. Oxford University Press, 2016.

[106] P. E. Black, M. Kass, K. Michael, and E. Fong, "NIST SP 500-268 v1.1. Source code security analysis tool functional specification version 1.1," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[107] R. W. Shirey, "Internet Security Glossary, Version 2." RFC 4949, Mar. 2013.

[108] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security Privacy*, vol. 3, pp. 81–84, Nov 2005.

[109] S. Shah and B. M. Mehtre, "An overview of vulnerability assessment and penetration testing techniques," *Journal of Computer Virology and Hacking Techniques*, vol. 11, pp. 27–49, February 2015.

[110] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, pp. 58–62, Mar. 2005.

[111] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[112] G. M. Weinberg and D. P. Freedman, "Reviews, walkthroughs, and inspections," *Software Engineering, IEEE Transactions on*, vol. SE-10, pp. 68–72, Jan 1984.

[113] W. S. Humphrey and M. I. Kellner, "Software process modeling: Principles of entity process models," in *Proceedings of the 11th International Conference on Software Engineering*, ICSE '89, (New York, NY, USA), pp. 331–342, ACM, 1989.

[114] E. Yourdon, *Structured Walkthroughs: 4th Edition*. Upper Saddle River, NJ, USA: Yourdon Press, 1989.

[115] B. Chess and J. West, "Prioritizing static analysis results," *Datenschutz und Datensicherheit - DuD*, vol. 34, no. 3, pp. 156–159, 2010.

[116] A. P. Moore, R. J. Ellison, and R. C. Linger, "Attack modeling for information security and survivability," 2001.

[117] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[118] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[119] C. J. V. Rijsbergen, *Information Retrieval*. Newton, MA, USA: Butterworth-Heinemann, 2nd ed., 1979.

[120] Y. Yang and X. Liu, "A re-examination of text categorization methods," in *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, (New York, NY, USA), pp. 42–49, ACM, 1999.

[121] "Software Assurance Reference Dataset Test Suites."

[122] M. Koo, R. Gaucher, C. Cleraux, and J. R. Rodriguez, "NIST SP 500-270 v1.1. Source code security analysis tool test plan version 1.1," tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[123] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, pp. 10:1–10:42, Mar. 2010.

[124] B. A. Kitchenham, I. C. Society, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and I. C. Society, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, pp. 721–734, 2002.

[125] J. E. McGrath, "Human-computer interaction," in *Readings in Human-Computer Interaction: Toward the Year 2000 (2nd ed* (R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg, eds.), ch. Methodology matters: doing research in the behavioral and social sciences, pp. 152–169, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995.

[126] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, pp. 135–137, Jan. 2001.

[127] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," tech. rep., National Institute of Standards and Technology, 2002.

[128] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, (New York, NY, USA), pp. 194–203, ACM, 2006.

[129] klog, "The frame pointer overwrite." `http://www.phrack.com/issues/55/8.html#article`.

Appendix A

Dataset

A.1 Adding Static Analysis Support for XSS

Support for Reflected XSS weaknesses in Dataset 2 and 3 specifically requires support for the cgic ANSI C library for CGI programming that was written by Thomas Boutell[1]. An example of this API can be seen in program 2198 within Dataset 2:

```
1  /* This software was developed at the National Institute of Standards and
2   * Technology by employees of the Federal Government in the course of their
3   * official duties. Pursuant to title 17 Section 105 of the United States
4   * Code this software is not subject to copyright protection and is in the
5   * public domain. NIST assumes no responsibility whatsoever for its use by
6   * other parties, and makes no guarantees, expressed or implied, about its
7   * quality, reliability, or any other characteristic.
8   *
9   * We would appreciate acknowledgement if the software is used.
10  * The SAMATE project website is: http://samate.nist.gov
11  */
12 #include <stdio.h>
13 #include <cgic.h>
14 #include <string.h>
15 #include <stdlib.h>
16
17 int cgiMain()
18 {
19     cgiHeaderContentType("text/html");
20     /*
```

[1] http://web.mit.edu/wwwdev/www/cgic.html and http://www.boutell.com/cgic/

```
21          Top of the page
22      */
23      fprintf(cgiOut, "<html><head>\n");
24      fprintf(cgiOut, "<title>Cross-Site Scripting: 1</title></head>\n");
25      fprintf(cgiOut, "<body><h1>XSS Test</h1>\n");
26      /*
27          If a the parameter 'q1','q2',etc. has some data, print it
28      */
29      char q[4][1024];
30      unsigned int i = 0;
31      for (; i < 4; ++i){
32          char name[4];
33          sprintf(name,"q%d",i);
34          cgiFormString(name, q[i], sizeof(q[i]));
35          if (strlen(q[i]))
36          {
37              fprintf(cgiOut, "Value number %d = %s<br />", i, q[i]);
38          }
39      }
40      /* Finish up the page */
41      fprintf(cgiOut, "</body></html>\n");
42      return 0;
43  }
```

In the code above, a new HTML page is being created on the fly and being written to `cgiOut`. The loop body performs a check to determine if the CGI script has been passed any arguments of the form *q1, q2, q3, or q4* and if present will write their associated values out to `cgiOut`, without any input validation, leading to a reflected XSS vulnerability. Upon further analysis, we note that the source of the weakness is the call to `cgiFormString` that does not perform any input validation prior to calling `fprintf` to write the retrieved argument to `cgiOut`.

To obtain strong evidence for XSS we should consider a data flow approach to static analysis. In HP Fortify SCA, a basic data flow requires at least two elements, a data flow source and a data flow sink [79]. A source rule needs to specify the target function and the taint that is to be tracked from that point. For example, both `cgiFormString` and `cgiFormStringNoNewLines` would be sources of taint in the cgic API.

```
1   <DataflowSourceRule formatVersion="3.2" language="cpp">
```

```
    <RuleID>15901788-53F2-4C7D-A3B4-AB914F77CDC9</RuleID>
    <TaintFlags>+XSS,+WEB</TaintFlags>
    <FunctionIdentifier>
      <NamespaceName>
        <Pattern/>
      </NamespaceName>
      <ClassName>
        <Pattern/>
      </ClassName>
      <FunctionName>
        <Pattern>cgiFormString|cgiFormStringNoNewlines</Pattern>
      </FunctionName>
      <ApplyTo implements="true" overrides="true" extends="true"/>
    </FunctionIdentifier>
    <OutArguments>1</OutArguments>
  </DataflowSourceRule>
```

The *XSS* taint that is marked by the source then needs to be tracked to the point where the tainted data is consumed in a potentially harmful manner (e.g. the call to `fprintf` above where the tainted value is being passed to any argument at index 2 or higher in the argument list of the function). While this rule will detect true positives when XSS taint hits `fprintf`, it will also potentially generate false positives if an XSS taint reaches an argument to `fprintf` for which the target file is not `cgiOut or standard out`. False positives could be reduced by writing a characterization sink rule instead (which enumerates the possible target files); however, this could lead to false negatives.

```
  <DataflowSinkRule formatVersion="3.16" language="cpp">
    <RuleID>FA43523C-4858-4A7A-8914-2217C4B6C7FC0</RuleID>
    <VulnKingdom>Input Validation and Representation</VulnKingdom>
    <VulnCategory>Cross-Site Scripting</VulnCategory>
    <VulnSubcategory>Reflected</VulnSubcategory>
    <DefaultSeverity>4.0</DefaultSeverity>
    <Description ref="desc.dataflow.cpp. cross_site_scripting_reflected ">
      <Explanation append="true"><![CDATA[This issue is being reported by a
          custom rule.]]></Explanation>
    </Description>
    <Sink>
      <InArguments>2...</InArguments>
```

```
12            <Conditional>
13                <And>
14                    <TaintFlagSet taintFlag="XSS"/>
15                    <Or>
16                        <TaintFlagSet taintFlag="WEB"/>
17                        <TaintFlagSet taintFlag="GUI_FORM"/>
18                        <TaintFlagSet taintFlag="FORM"/>
19                        <TaintFlagSet taintFlag="NETWORK"/>
20                    </Or>
21                    <Not>
22                        <TaintFlagSet
23                        taintFlag="VALIDATED_CROSS_SITE_SCRIPTING_REFLECTED"
24                        />
25                    </Not>
26                </And>
27            </Conditional>
28        </Sink>
29        <FunctionIdentifier>
30            <NamespaceName>
31                <Pattern/>
32            </NamespaceName>
33            <ClassName>
34                <Pattern/>
35            </ClassName>
36            <FunctionName>
37                <Pattern>fprintf</Pattern>
38            </FunctionName>
39            <ApplyTo implements="true" overrides="true" extends="true"/>
40        </FunctionIdentifier>
41    </DataflowSinkRule>
```

With these two rules above, and a few others, we can obtain support for XSS specific to the cgic API. While a given static analysis product may have support for a given type of weakness category, they may not support the specific semantics of a given API that contains the given weakness category.

A.2 Discussion of Dataset 3

During our design of this experiment, we conducted a review of the programs in Dataset 3 to ensure the accuracy of the code to the stated goals of the datasets. We identified 13 programs spanning five categories which contained exploitable vulnerabilities that violated the expected results for each program.

A.2.1 Command Injection

Four programs targeting Command Injection are in Dataset 3 that should not contain command injection weaknesses (2136, 2137, 2138, 2139). On detailed analysis, however, these appear to contain exploitable weaknesses as the added custom function (*purify*) that is intended to remove ";" is not sufficient to remove command injection vulnerabilities. Not reporting these cases as true positives would lead to a false sense of security. Alternatively, "if" `purify` was written to handle all the possible command injection special characters, a custom rule could be written for HP Fortify SCA to void the taint by adding the taint *VALIDATED_COMMAND_INJECTION* as a passthrough rule on `purify`. As a "security review" product, SCA's dataflow analyzer would then not report issues on the matching dataflow sinks since the *VALIDATED_COMMAND_INJECTION* would indicate that any potential malicious user-supplied input had been vetoed.

```
1  /* This software was developed at the National Institute of Standards and
2   * Technology by employees of the Federal Government in the course of their
3   * official duties. Pursuant to title 17 Section 105 of the United States
4   * Code this software is not subject to copyright protection and is in the
5   * public domain. NIST assumes no responsibility whatsoever for its use by
6   * other parties, and makes no guarantees, expressed or implied, about its
7   * quality, reliability, or any other characteristic.
8   *
9   * We would appreciate acknowledgement if the software is used.
10  * The SAMATE project website is: http://samate.nist.gov
11  */
12
13 #include <string.h>
14 #include <stdlib.h>
15 #include <stdio.h>
16 #include <stdbool.h>
```

```
17
18
19  #define SIZE_CMD 14
20  const char cmd[SIZE_CMD] = "/usr/bin/cat ";
21
22  /*
23          One of the most basic  filtering , remove the ';'
24  */
25  void purify(char *__buff)
26  {
27          char buf[512]="";
28          char *c = __buff, *b = buf;
29          for (;*c != '\0';c++)
30          {
31                  while (*c == ';') c++;
32                  *b++ = *c;
33          }
34          strncpy(__buff, buf, 512);
35  }
36
37
38  int main(int argc, char *argv[])
39  {
40          char sys[512];
41          char buff[512];
42
43
44          sys[0]= '\0';
45          if (fgets(buff,512 - SIZE_CMD,stdin))
46          {
47                  strcat(sys, cmd);
48                  strcat(sys, buff);
49                  purify(sys);
50                  switch(1==1){
51                          case true:
52                                  system(sys);
53                          default:
54                                  break;
55                  }
56          }
57          return 0;
58  }
```

The code above, from Dataset 3 case 2136, shows the retrieval of up to 512 bytes (less 14 bytes for the *cat command*) from standard input which is then concatenated to the end of the *cat* command to form the string that will be executed by the call to `system`. Recall that the `system` libc function executes a command by invoking */bin/sh -c command*. Prior to the invocation of `system` a call to the custom function `purify` has been added to avoid command injection weaknesses; however, this is insufficient protection for multiple reasons. First, simply removing malicious characters is not the best approach as it can lead to non-determinism. For example, if the program were passed *("/etc/passwd;/usr/bin/xterm")*, `purify` would turn this into an invalid path *("/etc/passwd/usr/bin/xterm")* when producing an error and responding accordingly would be more secure. Second, the function only protects against ";" separating multiple commands; however, bash shell commands can be separated by multiple other special symbols that will equally lead to command injection (e.g. semicolon (;), control operator (&&), a pipe (|), or command substitution(‘ or $()))[2]. An example of exploiting case 2136 can be seen in Figure A.1. All four programs in Dataset 3 relating to Command Injection have this coding pattern and as a result are true positives. Finally, a more secure approach would be to provide a white-listing approach where only a specific set of parameters are allowed by the program which are then compared at runtime using input validation.

Figure A.1: Experiment showing that programs that may report false positives actual contain exploitable vulnerabilities for Command Injection. Note the use of "&&" to separate commands rather than ";".

A.2.2 Cross-Site Scripting (XSS)

Two programs targetting XSS exist in Dataset 3 that should not contain exploitable XSS, however, upon detailed analysis they both contain exploitable reflected XSS

[2]Diaz *et al* [87] also noted the insufficient checks for command injection

(1920, 1924). Evaluation was conducted against cgic 2.05 and both the documentation and source code confirmed the weaknesses present in the code.

According to the documentation the `cgiHtmlEscape` function writes the output directly to `cgiOut` without modifying the original string and escaping only three problem characters[3]:

```
cgiFormResultType cgiHtmlEscape(char *s)

cgiHtmlEscape() outputs the specified null-terminated string to cgiOut, escaping any <, &,
and > characters encountered correctly so that they do not interfere with HTML markup. Returns
cgiFormSuccess, or cgiFormIO in the event of an I/O error.
```

In order to be protected from XSS in HTML many other protections should be considered. As a bare minimum six characters should be escaped (", ', and / are missing here)[4].

The documentation for `cgiHtmlEscape` is then confirmed by the source code in cgic.c where we note that `cgiHtmlEscape` invokes `cgiHtmlEscapeData` which then in turn invokes a macro to write the escaped characters directly to `cgiOut` via `putc`.

```
1   ...
2   #define TRYPUTC(ch) \
3           { \
4                   if (putc((ch), cgiOut) == EOF) { \
5                           return cgiFormIO; \
6                   } \
7           }
8
9   cgiFormResultType cgiHtmlEscapeData(char *data, int len)
10  {
11          while (len--) {
12                  if (*data == '<') {
13                          TRYPUTC('&');
14                          TRYPUTC('l');
15                          TRYPUTC('t');
16                          TRYPUTC(';');
17                  } else if (*data == '&') {
18                          TRYPUTC('&');
```

[3]http://www.boutell.com/cgic/#cgiHtmlEscape

[4]https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

```
19                              TRYPUTC('a');
20                              TRYPUTC('m');
21                              TRYPUTC('p');
22                              TRYPUTC(';');
23                      } else if (*data == '>') {
24                              TRYPUTC('&');
25                              TRYPUTC('g');
26                              TRYPUTC('t');
27                              TRYPUTC(';');
28                      } else {
29                              TRYPUTC(*data);
30                      }
31                      data++;
32              }
33              return cgiFormSuccess;
34      }
35
36      cgiFormResultType cgiHtmlEscape(char *s)
37      {
38              return cgiHtmlEscapeData(s, (int) strlen(s));
39      }
40      ...
```

Program 1920 has attempted to correct the true positive case from Dataset 2 by adding a call to `cgiHtmlEscape`, however, it appears the authors neglected to remove the following call to `fprintf` that still results in a reflected XSS vulnerability.

```
1       /* This software was developed at the National Institute of Standards and
2        * Technology by employees of the Federal Government in the course of their
3        * official duties. Pursuant to title 17 Section 105 of the United States
4        * Code this software is not subject to copyright protection and is in the
5        * public domain. NIST assumes no responsibility whatsoever for its use by
6        * other parties, and makes no guarantees, expressed or implied, about its
7        * quality, reliability, or any other characteristic.
8        *
9        * We would appreciate acknowledgement if the software is used.
10       * The SAMATE project website is: http://samate.nist.gov
11      */
12
13      #include <stdio.h>
14      #include <cgic.h>
```

```
15  #include <string.h>
16  #include <stdlib.h>
17
18
19  int cgiMain()
20  {
21          unsigned int false = 0 == 1;
22
23          cgiHeaderContentType("text/html");
24          /*
25                  Top of the page
26          */
27          fprintf(cgiOut, "<html><head>\n");
28          fprintf(cgiOut, "<title>Cross-Site Scripting: 1</title></head>\n");
29          fprintf(cgiOut, "<body><h1>XSS Test</h1>\n");
30          /*
31                  If a the parameter 'q' has some data, print it
32          */
33          char q[1][1024];
34          cgiFormString("q", q[0], sizeof(q[0]));
35          cgiHtmlEscape(q[0]);
36          fprintf(cgiOut, "Value = %s", q[0]);
37          /* Finish up the page */
38          fprintf(cgiOut, "</body></html>\n");
39          return 0;
40  }
```

The correct fix would have been to replace the call to `fprintf` with a call to `cgiHtmlEscape`. The same fix should be applied to program 1924.

```
1   /* This software was developed at the National Institute of Standards and
2    * Technology by employees of the Federal Government in the course of their
3    * official duties. Pursuant to title 17 Section 105 of the United States
4    * Code this software is not subject to copyright protection and is in the
5    * public domain. NIST assumes no responsibility whatsoever for its use by
6    * other parties, and makes no guarantees, expressed or implied, about its
7    * quality, reliability, or any other characteristic.
8    *
9    * We would appreciate acknowledgement if the software is used.
10   * The SAMATE project website is: http://samate.nist.gov
11   */
```

```
12
13  #include <stdio.h>
14  #include <cgic.h>
15  #include <string.h>
16  #include <stdlib.h>
17
18  void runCommand(const char *q)
19  {
20          fprintf(cgiOut, "Value = %s", q);
21  }
22
23
24  int cgiMain()
25  {
26          unsigned int false = 0 == 1;
27
28          cgiHeaderContentType("text/html");
29          /*
30                  Top of the page
31          */
32          fprintf(cgiOut, "<html><head>\n");
33          fprintf(cgiOut, "<title>Cross-Site Scripting: 1</title></head>\n");
34          fprintf(cgiOut, "<body><h1>XSS Test</h1>\n");
35          /*
36                  If a the parameter 'q' has some data, print it
37          */
38          char q[1024];
39          cgiFormString("q", q, sizeof(q));
40          cgiHtmlEscape(q);
41          runCommand(q);
42          /* Finish up the page */
43          fprintf(cgiOut, "</body></html>\n");
44          return 0;
45  }
```

Ensuring that this code is exploitable warrants a quick sanity check (which will later be verified using runtime monitoring). First compile the test project into an executable cgi script using a command similar to the following:

```
# gcc -Wall -g -O0 -o xss_scope_good.cgi xss_scope_good.c -I/opt/cgic205 -L/opt/cgic205 -lcgic
```

Finally, copy the cgi script into the necessary folder of the Web server and test the

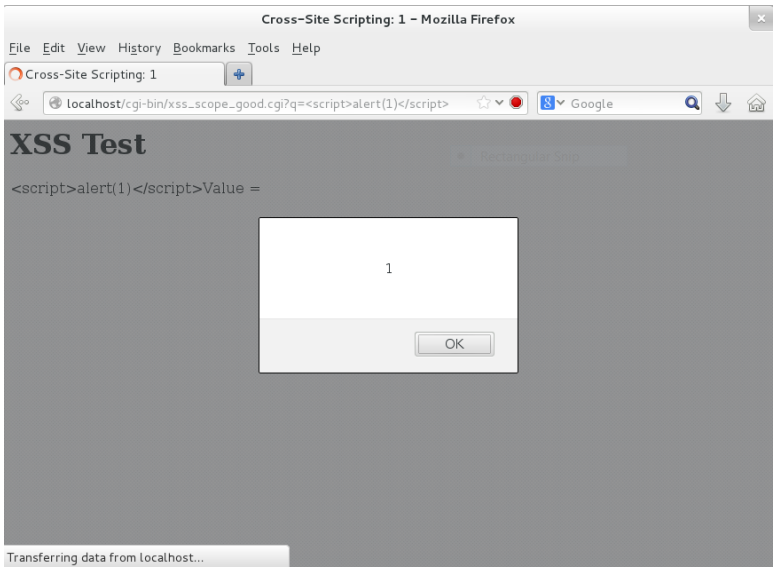

Figure A.2: Experiment showing that program intending to not have any XSS weaknesses contains an exploitable issue.

example in a browser. Using a URL such as the following:

"http://localhost/cgi-bin/xss_scope_good.cgi?q=
%3Cscript%3Ealert%281%29%3C/script%3E"

or

"http://localhost/cgi-bin/xss_scope_good.cgi?q=
hello;q1=%3Cscript%3Ealert%281%29%3C/script%3E"

would result is reflected XSS due to the reading of upto 1024 characters following parameter 'q' as shown in Figure A.2. The output on the HTML shown in the figure verifies our analysis that the `cgiHtmlEscape` function writes directly to `cgiOut` and the call to `fprintf` does result in reflected XSS.

```
# cp xss_scope_good.cgi /var/www/cgi-bin/
# service httpd start
```

A.2.3 Resource Injection

Four programs contain exploitable resource injection weaknesses due to a logic error in the code (1896, 1898, 1900, 1902). The present code will "only disallow" the reading

of files from the white-list rather than only allowing the reading of the white-list (in effect turning it into a black-list). For example, consider the code from 1898 and notice the return values are inverted for the conditional logic in function `allowed` compared to how it is called from `main` [5].

```
1  /* This software was developed at the National Institute of Standards and
2   * Technology by employees of the Federal Government in the course of their
3   * official duties. Pursuant to title 17 Section 105 of the United States
4   * Code this software is not subject to copyright protection and is in the
5   * public domain. NIST assumes no responsibility whatsoever for its use by
6   * other parties, and makes no guarantees, expressed or implied, about its
7   * quality, reliability, or any other characteristic.
8   *
9   * We would appreciate acknowledgement if the software is used.
10  * The SAMATE project website is: http://samate.nist.gov
11  */
12
13 #include <string.h>
14 #include <stdlib.h>
15 #include <stdio.h>
16
17
18 const char *whitelist[5] = {
19         "users_site.dat",
20         "users_reg.dat",
21         "users_info.dat",
22         "admin.dat",
23         "services.dat.cxx"
24 };
25
26
27 int allowed(const char *_str) {
28         for (unsigned i = 0; i < 5; i++)
29         {
30                 if (!strcmp(whitelist[i], _str))
31                         return 0;
32         }
33         return 1;
34 }
```

[5] Diaz *et al*[87] also noted the incorrect logic for resource injection in the programs

```
35
36
37  void printLine(const char *fileName)
38  {
39          FILE *fp = (FILE *)NULL;
40          if ((fp = fopen(fileName, "r")))
41          {
42                  char buff[512];
43                  if (fgets(buff, 512, fp))
44                  {
45                          printf("%s\n", buff);
46                  }
47                  fclose(fp);
48          }
49  }
50
51  int main(int argc, char *argv[])
52  {
53          if (argc > 1)
54          {
55                  if (allowed(argv[1]))
56                          printLine(argv[1]);
57          }
58          return 0;
59  }
```

A simple program to prove the vulnerability of this code can be seen in Figure A.3 by compiling the code and then passing the executable a file such as $\backslash etc \backslash passwd$:

```
# gcc -std=c99 -Wall -g -o resource_injection_basic_good resource_injection_basic_good.c
# ./resource_injection_basic_good /etc/passwd
```

```
[student@localhost 898]$ ./resource_injection_basic_good /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Figure A.3: Experimentation showing that a program written to not have Resource Injection weaknesses contains exploitable vulnerabilities.

If the programs were written as intended, and a false positive due to the presence of a custom white-list checking routine were present, then these can be suppressed in SCA by adding a simple custom passthrough rule on the white-list function that adds the VALIDATED_RESOURCE_INJECTION taint flag. The addition of such a rule

provides the semantics of the white-list function to the dataflow analyzer.

A.2.4 Heap Overflow

One program in the Heap Overflow vulnerability category still contained an overflow(1848). Specifically, the allocated buffer in the code below is overflowed by one byte.

```
1  /* This software was developed at the National Institute of Standards and
2   * Technology by employees of the Federal Government in the course of their
3   * official duties. Pursuant to title 17 Section 105 of the United States
4   * Code this software is not subject to copyright protection and is in the
5   * public domain. NIST assumes no responsibility whatsoever for its use by
6   * other parties, and makes no guarantees, expressed or implied, about its
7   * quality, reliability, or any other characteristic.
8   *
9   * We would appreciate acknowledgement if the software is used.
10  * The SAMATE project website is: http://samate.nist.gov
11  */
12
13
14 #include <stdlib.h>
15 #include <string.h>
16 #include <stdio.h>
17 #include <time.h>
18 // Binary Russian Dice
19
20 char *rand_text() {
21         srand(time(NULL));
22         unsigned length = (rand() % 25) - 1;
23         char *t = malloc(length * sizeof(char));
24         if (!t)
25                 return NULL;
26         unsigned i=0;
27         for (;i<length;++i)
28         {
29                 t[i] = (char)((rand() % 26) + 'a');
30         }
31         t[i+1] = '\0';
32         return t;
```

```
33  }
34
35  int main(int argc, char *argv[])
36  {
37          char *buf = (char *)NULL;
38          buf = malloc(25*sizeof(char));
39
40          if (buf != (char *)NULL)
41          {
42                  char *t = rand_text();
43                  if (t) {
44                          strcpy(buf,t);
45                          free(t);
46                  }
47                  free(buf);
48          }
49          return 0;
50  }
```

This program copies a randomly generated string, with a maximum length of 24, from one area of the heap to another buffer of size 25. The overflow starts in the `rand_text` function below when an area of memory is allocated from the heap using malloc. The size of the allocation, assuming a character size of one byte, will be in the range [-1,24] due to the nature of the modulus function and the behavior of the `rand` function. If `malloc` is passed a size of -1 then the return value will be *NULL* and error handling takes care of the rest. For all positive allocation sizes, however, we have a specialized form of buffer overflow known as an "off-by-one frame pointer overwrite" [129]. Even in the most limited overflow case, where only one byte past the end of a buffer can be manipulated, a successful exploit is possible. The program, however, is a bit different in that we cannot control what is written to the extra byte (it will always be NULL when *t[i+1]* is written) which limits the possible attack vectors further. The randomly generated buffer is then read in the body of `main` by `strcpy` which copies the contents of *t* into *buf* (recall that strcpy continues to copy bytes until the *NULL* byte is encountered resulting in an over-read). Ultimately, this code is not correct as the programmer could not have intended to place the null byte after the end of the allocated buffer (nor allocate memory of size -1).

A.2.5 Time-of-Check Time-of-Use

Both Time-of-Check Time-of-Use Race Condition programs still contain exploitable vulnerabilities (1892 and 1894). These race conditions arise when a window of opportunity is created between when a program checks a property of a resource prior to actually allocating the resource without proper synchronization locks in place. If the state or property of the resource changes after the check, but before the resource is allocated, then assumptions are invalid and undefined behavior can result. For example, the source code below from 1892 shows in the function `handler` an initial check to see if write permissions are available on the file.

```
1  /* This software was developed at the National Institute of Standards and
2   * Technology by employees of the Federal Government in the course of their
3   * official duties. Pursuant to title 17 Section 105 of the United States
4   * Code this software is not subject to copyright protection and is in the
5   * public domain. NIST assumes no responsibility whatsoever for its use by
6   * other parties, and makes no guarantees, expressed or implied, about its
7   * quality, reliability, or any other characteristic.
8   *
9   * We would appreciate acknowledgement if the software is used.
10  * The SAMATE project website is: http://samate.nist.gov
11  */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16 #include <sys/types.h>
17 #include <sys/wait.h>
18 #include <sys/stat.h>
19 #include <fcntl.h>
20 #include <errno.h>
21
22 const char fName[] = "test.file";
23 const char string [] = "What I want to write";
24 int fd;
25
26 void handler(int curPid)
27 {
28     struct flock lock;
29     if (access(fName, W_OK))
30         return;
```

```
31      fd = open(fName, O_WRONLY | O_CREAT | O_EXCL);
32      if (fd)
33      {
34          lock.l_type = F_WRLCK;
35          lock.l_whence = SEEK_SET;
36          lock.l_start = 0;
37          lock.l_len = 0;
38
39          if (fcntl (fd, F_SETLK, &lock) < 0) {
40              fprintf (stderr, "Can't lock %s\n", fName);
41              exit(EXIT_FAILURE);
42          }
43
44          fprintf (stdout, "(%d) Start handler...\n",curPid);
45          write(fd, (void *)string, sizeof(char) * 20);
46          close(fd);
47          fprintf (stdout, "(%d) Stop handler...\n",curPid);
48      }
49  }
50
51  int main(int argc, char *argv[])
52  {
53      pid_t pid = 0;
54      // create fork 1
55      if (fork())
56          return 0;
57
58      for (unsigned i=0;i<3;++i) {
59          pid = fork();
60          if (pid)
61          {
62              printf ("Run: %d\n",pid);
63              handler(pid);
64          }
65      }
66      return 0;
67  }
```

The fundamental problem with the above code is that `access` and `open` operate on a filename rather than a file descriptor which provides no guarantee that the file that access was checked against is the same file that was opened (since a change

could have occurred between the two calls). In addition to the call to `access` being vulnerable, the above code has other logic errors such as the parameters being passed to `open` contradicting the call to `access`. The combination of flags passed to `open` are assuming the file does not exist; however, we just called `access` which assumed the file existed.

Appendix B

Test Suite Improvements and Comparison

B.1 Accurate: Fixed 13 Incorrect Test Cases

Test suite 46 contains 13 test cases, discussed in Experiment 2, spanning five of 21 vulnerability categories, that do not fix the weakness they were designed to correct. Table 6.14 enumerates the specific programs, by test case ID and grouped by category, per weakness type. We contacted NIST and collaborated to correct the inconsistencies.

B.2 Precise: Removed Extraneous Weaknesses

Following the identification of 13 incorrect test cases in Test Suite 46 the code review collaboration with Aurelien Delaitre and Charles de Oliveira from NIST led to the identification of numerous extraneous weaknesses (defects of a different type than the test case is intended) in the test cases in both Test Suites 45 and 46. These weaknesses, along with minor code enhancements to fix business logic, have been removed to reduce noise generated by tools while analyzing the test suites (see details in Appendix C). These 112 extraneous weaknesses, summarized in Table B.1, are addressed and fixed in Test Suites 100 and 101.

Source Code Weakness	**CWE ID**	**TS45 & TS46 Programs (SARD Test Case ID)**
Improper Output Neutralization for Logs	117	93 1558 1559 1560
Stack-based Buffer Overflow	121	1797 1799 1801 1881 1883 1885 1930 2009(2x) 2136 2137 2138
Buffer Over-read	126	10 1831 1832
Improper Null Termination	170	10 1831 1832 1866 1868 1870 1895 1896 1899 1900
Uncleared Sensitive Information	226	1810 1835 1837 1839 1841 2130 2131 2132 2133
Unchecked Return Value	252	11 111 1573 1574 1646 1796 1797 1798 1799 1800 1801 1806 1807 1808 1809 1881 1883 1885 1914 1926 1930 2136 2137 2138 2139 2270
Incorrect Check of Function Return Value	253	1849 1850 1851 1854 1855 1856 2010 2012
Insufficient Entropy in PRNG	332	1828 1829 1830 1845 1848 1905 1906
Memory Leak	401	11 1737 1843 1844 2139
Double Free	415	1926
Use After Free	416	2200 (secondary)
Uninitialized Variable	457	1757 (secondary) 1881 1883 1885 2136 2137 2138 2195
Null Pointer Dereference	476	10 1931 2009 2134
Unused Variable	563	1588 1751 1875 1876 1877 1907 1908 1909 1910 2194 2195
Integer Overflow to Buffer Overflow	680	11 2139
Unchecked Return Value to Null Pointer Dereference	690	1737

Table B.1: Extraneous weaknesses found in *TS45* and *TS46*.

B.3 Consistent: Completed GOOD/BAD Pairs

In order to improve consistent measurement we needed to ensure that every bad test case has an equivalent good one. Our investigation discovered that 16 pairs could be identified as indirect good/bad pairs, 15 test cases were identified as existing for pairs in SARD but not in the test suite, 27 test cases were created to establish a good/bad pair where none existed, 37 test cases had existing pairs, and 2 test case pairs were created from scratch. The result is 96 GOOD/BAD pairs in the new test suites (as can be seen in the Table B.2 at the end of this appendix).

B.4 Automation and Metadata Enhancements

While improving the accuracy, precision, and consistency were a primary focus, we also endeavored to make the test suites more consumable by researchers by providing consistent naming conventions, labeling, and metadata.

For automation, file names were updated with a suffix marking each test case as either “-good” or “-bad”. Additionally, each test case was marked with comments to indicate the line where either “Flaw” or “Fix” was located.

Our revisions to SARD also enhance the metadata associated with individual test programs. First, the new test suites had a positive impact on the NIST SARD, through the addition of a new feature called the 'Association' field. The new section allows users to navigate between good/bad test case pairs, explore replaced/replacing hierarchies of deprecated test cases, and jump to suites to which test cases belong.

We also adopted a systematic approach to describing each new test case with its specific characteristics for metadata information. Using descriptions from *TS45* and *46* test cases as input, we passed the description content through several revisions to ensure correctness and consistency prior to addition to *TS100* and *TS101*. Test cases that did not have a replacement pair (good/bad) in *TS45* and *TS46* derive their description from the negation of their good/bad pair.

During our review process we found that compiling and scanning *TS45* and *TS46* produced a list of test cases with incorrect syntax and missing library dependencies. In addition to numerous test cases not specifying their necessary includes, test case 2139 had issues related to its implementation. The issues in test case 2139 included the following: i) missing type specifier for variable `buffLength` on line 21; ii) typo in `buffLength` on line 27; iii) wrong attribution to `buf` on line 34; iv) missing semi-colon at the end of line 35; and v) the missing underscore for `__buff` on line 38. SQL Injection test cases depend on *mysqlclient* library and textitmysql.h header in order to compile correctly. Finally, Cross-Site Scripting test cases also require the CGIC library. Such compile-time details may cause inconsistent results for different researchers. To help ensure that researchers obtain comparable results, compiler commands were included in the *Instructions* metadata section of test cases of *TS100* and *TS101*.

Upgrading and replacing *TS45* and *TS46* led to the deprecation of test cases from these suites. Individual notes were added in the *Comments* section for test cases that have been replaced in the SARD by *TS100* and *TS101*. Each note provides the main reason why the test case was replaced and directs users to navigate between replaced/new test cases via the *Association* metadata field (see Table B.2).

Authorship and copyright of test cases from *TS45* and *TS46* is attributed to Michael Koo, Romain Gaucher, and Fortify, among others. Accurate attribution of authorship is preserved in *TS100* and *TS101*, by importing all authors from the original test cases and duplicating authorship when a good/bad pair was created. In addition, the original authors' are complemented by adding the names of individuals who were responsible for the improvements to the source code. In some test cases, which originated from *TS45* and *TS46*, NIST is also listed as an author.

Test cases 1794, 1795, 1919-1924, 2198 and 2204 from *TS45* and *TS46* are categorized as *CWE-79: Improper Neutralization of Input During Web Page Generation (XSS)*, creating a statement that accepts a user input from an HTML page variable *q*, then printing it out without a specific sanitizing filtering. This weakness' documentation covers different types of XSS, one of them being *"Reflected XSS (or Non-Persistent)"*, representing the test cases above. *CWE-80: Basic XSS*, specializes CWE-79 and therefore we updated the metadata in the replacement test cases 149093, 149094, 149173-149178, 149215, and 149216 respectively in *TS100* and *TS101*. The same approach was used for test case 149054 from *TS101* replacing 1646. The older version implements a *CWE-020: Improper Input Validation* while the new one carries similar source code, but a more specific weakness category as *CWE-78: OS Command Injection*.

In Section B.2 we corrected test cases 1806-1809 as having extraneous weaknesses Furthermore, test cases 1807 and 1809 were identified as *CWE-362: Race Condition*, while their bad pairs 1806 and 1808, respectively, implemented *CWE-367: TOCTOU*. The new test cases 149102 and 149104 also fix the CWE mismatch.

B.5 Summary:

The last decade has seen great strides towards improving the maturity of software security assurance through both techniques and the ability to measure results. Test suites 100 and 101 are now available as of April 29th, 2015, providing numerous improvements including the removal of targeted weaknesses in 13 test cases from *TS46*, removal of extraneous weaknesses in test cases from *TS45* and *TS46*, addition of missing test cases for GOOD/BAD pairings across test suites, code updates for quality and automation, removal of bugs/"incorrect logic" (e.g. forking), updating of file names to indicate "good" and "bad" test cases, and marking of "Flaw" and "Fix" locations in the code. The new test cases also provide improved metadata on the SARD website including updated "replaces"/"replaced-by" mappings to improve associations, updated "pair" mappings to improve associations, updated descriptions, inclusion of compiler instructions for consistency of research results, notes explaining deprecations, and corrected CWE mappings.

Defining multiple input strings for every test case in SARD would also be beneficial to software security assurance activities. Specifically, if a test case contains a specific weakness, then example permutations that exploit the weakness would be beneficial.

For example, for command injection, input strings demonstrating an exploit for each special character that can exploit the weakness would expose the latent defect (recall that the `purify` function only removed ';').

After a great deal of effort these revised test suites still do not meet all of the requirements that were listed in the appendix of the specification and test plan documents since the specification lists 17 complexities, each of which has multiple permutations, across 21 CWEs [106, 122]. Continuous improvements to software security assurance techniques and measurement are still needed. As part of the investigation into the adequacy of test suites 45 and 46 for the purpose of evaluation, a gap analysis was conducted to determine how complete the test suite was in relation to the stated requirements in the test plan. The findings and associated questions were communicated to NIST on March 30th, 2015. NIST decided to not update the test plan nor create a new test plan at this time to address the noted deficiencies and would use the feedback in the future when the creation of a new test plan became a priority. The results of this investigation are included in Appendix D.

For example, ignoring the permutations of enumerations for each complexity, absolute minimal coverage would require 130 unique test cases for evaluating true positives for each complexity over the 21 CWEs. Furthermore, if we assume that all of the complexities and their enumerations are complete, and that combinations of permutations involving different enumerations from different complexities are not considered, we would require at least 610 unique test cases just to cover one test case for each complexity and their enumerations defined in the appendix of NIST SP 500-268 v1.1.

Even with all the improvements, the test suites do not satisfy all of the relevant complexities listed in the specification. The newly added test suites 100 and 101 are divided into two parts:

- 100: defines test cases that are vulnerable to stated weakness and is useful for determining false negatives;
- 101: defines test cases that are not vulnerable to stated weakness and is useful for determining false positives;

These new test suites do not cover all of the relevant complexities for each CWE nor the relevant enumerations for each complexity (for each CWE). The test suites are useful for determining if a software security assurance product is capable of finding these specific issues; however, it is not sufficient for evaluating the true capabilities nor coverage of a specific product.

Finally, the following table enumerates the test cases in the original *TS45* and *TS46*, as included in the downloaded zip files from SARD, versus the replacement test suites *TS100* and *TS101*. Labeling indicates the reasons for deprecating old test cases and creating new test cases. Descriptions A→D only apply to old test cases prior to *TS100* and *TS101*. Descriptions E and F only apply to new test cases included in *TS100* and *TS101*.

A: This test case is deprecated due to renaming of files and the addition of FLAW/FIX comments to assist automation. A new test case has been added. Please refer to the "Association" field to view the replacement test case.

B: This test case is deprecated due to minor edits to align with BAD/GOOD pairing, improvements to code, renaming of files, and the addition of FLAW/FIX comments to assist automation. A new test case has been added. Please refer to the "Association" field to view the replacement test case.

C: This test case is deprecated due to removal of extraneous weaknesses, improvements to code, renaming of files, and the addition of FLAW/FIX comments to assist automation. A new test case has been added. Please refer to the "Association" field to view the replacement test case.

D: This test case is deprecated due to the fixing of the CWE this test case is not supposed to contain, removal of extraneous weaknesses, renaming of files, and the addition of FLAW/FIX comments to assist automation. A new test case has been added. Please refer to the "Association" field to view the replacement test case.

E: This test case was added in order to create a GOOD/BAD pair for the previously existing test case [TESTCASE ID]. Please refer to the "Association" field to view the valid pair and the test case it replaced.

F: This test case was added in order to provide a valid test case under [CWE-ID].

−: Test case does not exist in SARD.

[ID]: A test case that forms a pair, where the test case exists in SARD but not in the test suites.

$ID^{'}$: An indirect good/bad pairing, consisting of two test cases that were in *TS45* and *TS46*. We call these indirect since they are not, by default, paired. It is the familial replaces/replaced hierarchy creating the good/bad pairing, similar to how a first-cousin, once removed, is considered an indirect cousin within family trees.

ID^{δ}: Test case replaced due to incorrect CWE.

ID^{α} Marks a test case that was removed as a duplicate.

Category	**Source Code Weakness**	**CWE ID**	Vulnerable Programs (SARD Test Case ID)		Fixed Programs (SARD Test Case ID)	
			TS45	**TS100**	**TS46**	**TS101**
Input Validation	OS Command Injection	78	111 *C*	149053	[1646] *C*	149054
			1881′ *C*	149151	2136′ *D*	149152
			1883′ *C*	149153	2137′ *D*	149154
			1885′ *C*	149155	2138′ *D*	149156
			[11] *C*	149241	2139 *D*	149242
	Basic XSS	80	1781′ *B*	149177	1924′ *D*	149178
			1794 *B*	149093	1795 *A*	149094
			1919 *B*	149173	1920 *D*	149174
			1921 *B*	149175	1922 *B*	149176
			2198′ *A*	149215	2204′ *A*	149216
	SQL Injection	89	1796 *C*	149095	1797 *C*	149096
			1798 *C*	149097	1799 *C*	149098
			1800 *C*	149099	1801 *C*	149100
			-	149187 *E*	1930 *C*	149188
	Resource Injection	99	1895 *C*	149157	1896 *D*	149158
			1897 *B*	149159	1898 *D*	149160
			1899 *C*	149161	1900 *D*	149162
			1901 *B*	149163	1902 *D*	149164
Range Errors	Stack Overflow	121	1544 *A*	149055	1545 *A*	149056
			[1546] *A*	149057	1547 *A*	149058
			1548 *A*	149059	1549 *A*	149060
			1563 *A*	149065	[1564] *A*	149066
			1565 *A*	149067	1566 *A*	149068
			[1601] *A*	149077	1602 *A*	149078
			1751 *C*	149087	-	149088 *E*
			1905 *C*	149165	1906 *C*	149166
			1907 *C*	149167	1908 *C*	149168
			1909 *C*	149169	1910 *C*	149170
			2009 *C*	149193	-	149194 *E*
	Heap Overflow	122	[1573] *C*	149069	1574 *C*	149070
			1611 *A*	149079	-	149080 *E*
			1612 *A*	149081	1613 *A*	149082
			[1614] *A*	149083	1615 *A*	149084
			1843 *C*	149123	1844 *C*	149124
			1845′ *C*	149125	1848′ *D*	149126
			[15] *C*	149201	2134 *C*	149202
	Format String Vulnerability	134	10 *C*	149237	-	149238 *E*
			92′ *A*	149045	1556′ *A*	149046
			93 *C*	149047	[1558] *C*	149048
			[1559] *C*	149061	1560 *C*	149062
			-	149063 *E*	1562 *A*	149064
			1831 *C*	149111	1832 *C*	149112
			1833 *A*	149113	1834 *A*	149114
	Improper Null Termination	170	1849′ *C*	149127	1856′ *C*	149128
			1850 *C*	149129	[1851] *C*	149130
			1854 *C*	149131	1855 *C*	149132
			1857 *B*	149133	1858 *B*	149134

			2010′ C	149195	2012′ C	149196
API Abuse	Heap Inspection	244	1737 C	149085	-	149086 E
	Often Misused String Management	251	1865 A	149137	1866 C	149138
			1867 A	149139	1868 C	149140
			1869 A	149141	1870 C	149142
			1871 A	149143	1872 B	149144
			1873 A	149145	1874 B	149146
Security Features	Hard-coded Password	259	1810 C	149105	-	149106 E
			1835′ C	149115	2130′ C	149116
			1837′ C	149117	2131′ C	149118
			1839′ C	149119	2132′ C	149120
			1841′ C	149121	2133′ C	149122
Time and State	Time-of-check Time-of-use Race Condition	367	102 B	149051	-	149052 E
			1806 C	149101	[1807] C	149102
			1808 C	149103	[1809] C	149104
					1892^{δ} D	
					1894^{δ} D	
			-	149231 F	-	149232^{δ} F
			-	149233 F	-	149234^{δ} F
	Unchecked Error Condition	391	1928 A	149185	1929 A	149186
Code Quality	Memory Leak	401	1585 A	149071	1586 A	149072
			1588 C	149073	1589 A	149074
			-	149179 E	1925 B	149180
			-	149181 E	1926 C	149182
			-	149189 E	1933 B	149190
	Unrestricted Critical Resource Lock	412	2109′ A	149199	2205′ B	149200
	Double Free	415	99 A	149049	-	149050 E
			1446^{α}			
			1590 A	149075	1591 A	149076
			1827 B	149107	1828 C	149108
			1829 C	149109	1830 C	149110
			2199 A	149217	-	149218 E
			-	149229 E	2271 A	149230
	Use After Free	416	[1913] A	149239	1914 A	149240
			-	149203 E	2135 B	149204
			2108^{α}			
			2200 C	149219	-	149220 E
			2201′ A	149221	2269′ A	149222
			2202 B	149223	-	149224 E
			2203 B	149225	-	149226 E
			[1911] B	149247	2270 A	149248
	Uninitialized Variable	457	1757 C	149089	-	149090 E
			2003 A	149191	-	149192 E
			2019 B	149197	-	149198 E
			-	149207 E	2186 B	149208
	Unintentional Pointer Scaling	468	1782 B	149091	-	149092 E
			-	149183 E	1927 B	149184
	Null Dereference	476	1875 C	149147	1876 C	149148
			1879 A	149149	1880 A	149150
			2193 B	149209	-	149210 E
			1877′ B	149243	2194′ C	149244

			-	149245 *E*	2195 *C*	149246
Encapsulation	Leftover Debug Code	489	1861 *B*	149135	1862 *B*	149136

Table B.2: Weakness categories covered by *TS100* and *TS101* (Replace *TS45* and *TS46*).

Appendix C

Extraneous Weaknesses

Through a collaboration with Aurelien Delaitre and Charles de Oliveira, from NIST, the following extraneous weaknesses were also identified and resolved as part of our work to improve and replace test suites 45 and 46 and presented at EASE 2016 [29].

Test cases 111/1646 for *CWE-78: OS Command Injection* disregard an important function's return value (*CWE-252: Unchecked Return Value*). Test case 149053/ 149054, their replacements, handle the function's result accordingly.

Test cases 1881/2136, 1883/2137 and 1885/2138, implementing the same weakness, could trigger a buffer overflow when concatenating a string to an uninitialized buffer (*CWE-457: Uninitialized Variable* as well as *CWE-121: Stack-based Buffer Overflow*). This problem was fixed by replacing the concatenation operation by a simple string copy, initializing the buffer at the same time. These test cases also fail to check the return value of an important function (*CWE-252: Unchecked Return Value*). Their replacements (149151-149156) fix the problem.

Test cases 11/2139 for *CWE-78: OS Command Injection* calculate a buffer size incorrectly that leads to a buffer overflow (*CWE-680: Integer Overflow to Buffer Overflow*). They also fail to check the return value of an important system function (*CWE-252: Unchecked Return Value*) creating a potential memory leak (*CWE-401: Memory Leak*). Replacements 149241/149242 fix these defects.

In test cases 1796-1801 and 1930 for *CWE-89: SQL Injection*, the database initialization function might fail, but the return value is not checked (*CWE-252: Unchecked Return Value*), leading to unexpected behavior. Their replacements (149095-149100 and 149188) check return values. In addition, good test cases 1797, 1799, 1801 and 1930 use an encoding function to prevent injection. However, they do not allocate sufficient memory to store the encoded string, leading to *CWE-121: Stack-based Buffer*

Overflow. Their replacements calculate the buffer size correctly.

Test cases 1895/1896 and 1899/1900 for *CWE-99: Resource Injection* do not ensure buffer null-termination (*CWE-170: Improper Null Termination*). Replacement test cases 149157/149058 and 149161/149162 ensure termination.

Test cases 2009 (*CWE-121: Stack-based Buffer Overflow*) and 2134 (*CWE-122: Heap-based Buffer Overflow*) access their program arguments without verifying existence first, potentially causing *CWE-476: NULL Pointer Dereference*. Replacements 149193/149194 and 149201/149202 ensure parameters existence. Additionally, the three sites triggering a buffer overflow in test case 2009 have been reduced to one.

Test cases 1573/1574, for *CWE-122: Heap-based Buffer Overflow*, do not check the return value of a number processing function (*CWE-252: Unchecked Return Value*), leading to unexpected behavior. Replacement test cases 149069/149070 ensure the operation succeeded or aborts.

Test cases 1843/1844 implementing *CWE-122: Heap-based Buffer Overflow* allocate memory but do not free it on all paths, leading to *CWE-401: Memory Leak*. Replacements 149123/149124 ensure the memory is freed properly.

Test cases 93/1558 and 1559/1560 for *CWE-134: Uncontrolled Format String* both contain *CWE-117: Improper Output Neutralization for Logs* vulnerabilities. Replacements (149047/149048, 1149061/149062) implement a whitelist filter to prevent any kind of injection.

Test case 10 for *CWE-134: Uncontrolled Format String* reads a large amount of memory from a parameter regardless of the latter's size. This leads to *CWE-126: Buffer Over-read* and *CWE-170: Improper Null Termination* that replacement test case 149237 fixes by using a string copy and adding a null terminator. Also, this test case, along with 1931, uses the program's arguments without checking their number, potentially leading to *CWE-476: NULL Pointer Dereference*. Replacements validate arguments. Test cases 1831/1832, for the same CWE, do not always null-terminate a string (*CWE-170: Improper Null Termination*) leading to *CWE-126: Buffer Over-read*. Replacement test cases 149111/149112 systematically add a null terminator.

Test cases 1849/1856, 1850/1851, 1854/1855 and 2010/2012 for *CWE-170: Improper Null Termination* do not check the return value of reading functions properly (*CWE-253: Incorrect Check of Function Return Value*). Test cases 149127-149132 and 149195/149196 correct the defect.

Test case 1737 for *CWE-244: Heap Inspection* does not check if a memory reallocation function fails, leading to *CWE-690: Unchecked Return Value to NULL Pointer*

Dereference. Also, one buffer is not freed on all paths (*CWE-401: Memory Leak*). Replacement 149085 corrects both issues.

Test cases 1866, 1868 and 1870 for *CWE-251: Misused String Management* misuse a string copy function by failing to set a null terminator at the end of the destination buffer, causing *CWE-170: Improper Null Termination*. Replacements 149138, 49140 and 149142, add the terminator.

Test cases 1810, 1835/2130, 1837/2131, 1839/2132 as well as 1841/2133 for *CWE-259: Hard-coded Password* prompt the user for a password but fail to erase it before release, leading to *CWE-226: Uncleared Sensitive Information*. Their replacements (149105, 149115-149122) implement secure functions to acquire the user's password and erase it.

Test cases 1806-1809 for *CWE-367: TOCTOU Race Condition* fail to check the return value of writing functions (*CWE-252: Unchecked Return Value*). Test cases 149101-149104 fix the problem.

Test case 1926 for *CWE-401: Memory Leak* fails to check for successful memory allocation (*CWE-252: Unchecked Return Value*). Replacement 149182 does. This defect can also lead to *CWE-415: Double Free*. Test cases 1914 and 2270 also fail to check the return value of memory allocation. Replacements 149240 and 149248 do.

Test case 2200 for *CWE-416: Use After Free* contains two sites for this weakness. The second site has been removed in replacement test case 149219.

Test case 1757 for *CWE-457: Uninitialized Variable* bears two sites for that defect. Test case 149089 merges the sites.

Test case 1588 for *CWE-401: Memory Leak* assigns a value to a variable that is never read, leading to *CWE-563: Unused Variable*. Replacement 149073 prints the value to ensure it is read after assignment. In addition, 1875/1876, 1877/2194 and 2195 for *CWE-476: NULL Pointer Dereference* and test cases 1751 and 1907-1910 for *CWE-121: Stack-based Buffer Overflow* contain the same error (CWE-563). The replacement test cases 149147/149148, 149243/149244, 149246, 149087 and 149167-149170 implement the same fix. Test case 2195 also references a potentially uninitialized argument *CWE-457*. Replacement 149246 fixes this by initializing the local variable to null rather than the indexed argument.

Finally, test cases 1905/1906, 1845/1848 and 1828-1830 use a weak random number generator (*CWE-332: Insufficient Entropy in PRNG*). The problem is fixed in test cases 149165/149166, 149125/149126 and 149108-149110 by using system-provided entropy-based random numbers.

Appendix D

Further Test Suite Improvements

D.0.1 Gap Analysis of Complexity Coverage

Results of the gap analysis, comparing the test suite requirements against the actual test cases included in test suites 45 and 46, are depicted in Table D.1 where the highlighted items represent inconsistencies from the test plan.

CWE ID	**Test Suite 45 (Spec. Labeled) reordered**	**Test Suite 46 (Spec. Labeled) reordered**	**Test Suite 100**	**Test Suite 101**	**Relevant Complexities Covered? (From Spec.)**
	111 (Basic)		149053 (Basic)	149054 (Basic)	Taint
		2139 (Basic)	149241 (Basic)	149242 (Basic)	
	1885 (Scope)	2138 (Scope)	149155 (Scope)	149156 (Scope)	Scope
					AddrAlias
78					Container
	1881 (LContFlow)	2136 (LContFlow)	149151 (LContFlow)	149152 (LContFlow)	LContFlow
	1883 (LoopStr)	2137 (LoopStr)	149153 (LoopStr)	149154 (LoopStr)	LoopStr
					BuffAddrT
	1794 (Basic)	1785 (Basic)	149093 (Basic)	149094 (Basic)	Taint
	1781 (Scope)	1924 (Scope)	149177 (Scope)	149178 (Scope)	Scope
	1919 (AddrAlias)	1920 (AddrAlias)	149173 (AddrAlias)	149174 (AddrAlias)	AddrAlias
80/*79*	1921 (Container)	1922 (Container)	149175 (Container)	149176 (Container)	Container
					LContFlow
	2198 (LoopComp)	2204 (LoopComp)	149215 (LoopComp)	149216 (LoopComp)	LoopStr
					BuffAddrT
	1796 (Basic)	1797 (Basic)	149095 (Basic)	149096 (Basic)	Taint
	1798 (ArrIndxC)	1799 (ArrIndxC)	149097 (ArrIndxC)	149098 (ArrIndxC)	
	1800 (Scope)	1801 (Scope)	149099 (Scope)	149100 (Scope)	Scope
					AddrAlias
89					Container
					LContFlow
		1930 (LoopStr)	149187 (LoopStr)	149188 (LoopStr)	LoopStr
					BuffAddrT

	1897 (Basic)	1898 (Basic)	149159 (Basic)	149160 (Basic)	Taint
	1901 (Scope)	1902 (Scope)	149163 (Scope)	149164 (Scope)	Scope
	1895 (AddrAlias)	1896 (AddrAlias)	149157 (AddrAlias)	149158 (AddrAlias)	AddrAlias
99	1899 (Container)	1900 (Container)	149161 (Container)	149162 (Container)	Container
					LContFlow
					LoopStr
					BuffAddrT
	2009 (Basic)		149193 (Basic)	149194 (Basic)	Taint
		1547 (Basic)	149057 (Basic)	149058 (Basic)	
	1563 (Basic)		149065 (Basic)	149066 (Basic)	
	1565 (Basic)	1566 (Basic)	149067 (Basic)	149068 (Basic)	
		1602 (Basic)	149077 (Basic)	149078 (Basic)	
	1548 (Scope)	1549 (Scope)	149059 (Scope)	149060 (Scope)	Scope
					MemLoc
					MemAcc
	1909 (LoopStr)	1910 (LoopStr)	149169 (LoopStr)	149170 (LoopStr)	LoopStr
					LoopIt
					LoopComp
121					LContFlow
	1907 (IndexAli)	1908 (IndexAli)	149167 (IndexAli)	149168 (IndexAli)	IndexAli
					DataType
					Container
					BuffAddrT
					Async
	1905 (ArrLenC)	1906 (ArrLenC)	149165 (ArrLenC)	149166 (ArrLenC)	ArrLenC
	1544 (ArrIndxC)	1545 (ArrIndxC)	149055 (ArrIndxC)	149056 (ArrIndxC)	ArrIndxC
	1751 (ArrIndxC)		149087 (ArrIndxC)	149088 (ArrIndxC)	
					ArrAddrC
					AddrAlias
	1611 (Basic)		149079 (Basic)	149080 (Basic)	Taint
		2134 (Basic)	149201 (Basic)	149202 (Basic)	
	1612 (Scope)	1613 (Scope)	149081 (Scope)	149082 (Scope)	Scope
		1615 (Scope)	149083 (Scope)	149084 (Scope)	
	1845 (ArrIndxC)	1848 (MemLoc)	149125 (MemLoc)	149126 (MemLoc)	MemLoc
					MemAcc
					LoopStr
					LoopIt
					LoopComp
122					LContFlow
					IndexAli
					DataType
					Container
					BuffAddrT
					Async
					ArrLenC
					ArrIndxC
	1843 (ArrAddrC)	1844 (ArrAddrC)	149123 (ArrAddrC)	149124 (ArrAddrC)	ArrAddrC
		1574 (ArrAddrC)	149069 (ArrAddrC)	149070 (ArrAddrC)	
					AddrAlias
	10 (Basic)		149237 (Basic)	149238 (Basic)	Taint
	92 (AddrAlias)	1556 (Scope)	149045 (Scope)	149046 (Scope)	Scope
	93 (Scope)		149047 (Scope)	149048 (Scope)	
		1562 (Scope)	149063 (Scope)	149064 (Scope)	
134					

		1560 (AddrAlias)	149061 (AddrAlias)	149062 (AddrAlias)	AddrAlias
	1831 (Container)	1832 (Container)	149111 (Container)	149112 (Container)	Container
	1833 (LContFlow)	1834 (LContFlow)	149113 (LContFlow)	149114 (LContFlow)	LContFlow
					LoopStr
					BuffAddrT
170	1849 (Basic)	1856 (Basic)	149127 (Basic)	149128 (Basic)	Taint
	1857 (Taint)	1858 (Taint)	149133 (Taint)	149134 (Taint)	
					Scope
	1850 (AddrAlias)		149129 (AddrAlias)	149130 (AddrAlias)	AddrAlias
	1854 (Container)	1855 (Container)	149131 (Container)	149132 (Container)	Container
					LContFlow
					LoopStr
	2010 (BuffAddrT)	2012 (BuffAddrT)	149195 (BuffAddrT)	149196 (BuffAddrT)	BuffAddrT
244	1737 (Basic)		149085 (Basic)	149086 (Basic)	Taint
					Scope
					AddrAlias
					Container
					LContFlow
					LoopStr
					BuffAddrT
251	1865 (Basic)	1866 (Basic)	149137 (Basic)	149138 (Basic)	Taint
	1873 (Taint)	1874 (Taint)	194145 (Taint)	149146 (Taint)	
	1871 (Scope)	1872 (Scope)	149143 (Scope)	149144 (Scope)	Scope
	1867 (AddrAlias)	1868 (AddrAlias)	149139 (AddrAlias)	149140 (AddrAlias)	AddrAlias
	1869 (Container)	1870 (Container)	149141 (Container)	149142 (Container)	Container
					LContFlow
					LoopStr
					BuffAddrT
259	1810 (Basic)		149105 (Basic)	149106 (Basic)	Scope
					AddrAlias
	1837 (Container)	2131 (Basic)	149117 (Container)	149118 (Container)	Container
	1839 (LContFlow)	2132 (LContFlow)	149119 (LContFlow)	149120 (LContFlow)	LContFlow
	1841 (LoopStr)	2133 (LoopStr)	149121 (LoopStr)	149122 (LoopStr)	LoopStr
					BuffAddrT
	1835 (ArrAddrC)	2130 (ArrAddrC)	149115 (ArrAddrC)	149116 (ArrAddrC)	
367	102 (Basic)		149051 (Basic)	149052 (Basic)	
	1806 (Basic)		149101 (Basic)	149102 (Basic)	
		1892 (Basic)	149231 (Basic)	149232 (Basic)	
	1808 LContFlow		149103 (LContFlow)	149104 (LContFlow)	
		1894 (LContFlow)	149231 (LContFlow)	149232 (LContFlow)	
					asynchronous
391	1928 (Basic)	1929 (Basic)	149185 (Basic)	149186 (Basic)	
401	1585 (Basic)	1933 (Basic)	149189 (Basic)	149190 (Basic)	
	1588 (Scope)	1586 (Scope)	149071 (Scope)	149072 (Scope)	Scope
		1589 (AddrAlias)	149073 (AddrAlias)	149074 (AddrAlias)	AddrAlias
		1925 (Container)	149179 (Container)	149180 (Container)	Container
					LContFlow
		1926 (LoopStr)	149181 (LoopStr)	149182 (LoopStr)	LoopStr
412	2109 (Basic)	2205 (Basic)	149199 (Basic)	149200 (Basic)	
					asynchronous
	2199 (Basic)		149217 (Basic)	149218 (Basic)	
		2271 (Basic)	149229 (Basic)	149230 (Basic)	

	1590	(Scope)	1591	(Scope)	149075	(Scope)	149076	(Scope)	Scope
									AddrAlias
									Container
	1827	(LContFlow)	1828	(LContFlow)	149107	(LContFlow)	149108	(LContFlow)	LContFlow
	1829	(LoopStr)	1830	(LoopStr)	149109	(LoopStr)	149110	(LoopStr)	LoopStr
	99	(BuffAddrS)			149049	(BuffAddrS)	149050	(BuffAddrS)	BuffAddrT
416	2200	(Basic)			149219	(Basic)	149220	(Basic)	
	2201	(Scope)	2269	(Scope)	149221	(Scope)	149222	(Scope)	Scope
			2270	(AddrAlias)	149247	(AddrAlias)	149248	(AddrAlias)	AddrAlias
	2202	(Container)			149223	(Container)	149224	(Container)	Container
			2135	(Container)	149203	(Container)	149204	(Container)	
									LContFlow
									LoopStr
	2203	(BuffAddrT)			149225	(BuffAddrT)	149226	(BuffAddrT)	BuffAddrT
			1914	(BuffAddrT)	149239	(BuffAddrT)	149240	(BuffAddrT)	
457			2186	(Basic)	149207	(Basic)	149208	(Basic)	
	2019	(Datatype)			149197	(Datatype)	149198	(Datatype)	
	2003	(Datatype)			149191	(DataType)	149192	(Datatype)	
									Scope
									AddrAlias
									Container
									LContFlow
	1757	(LoopStr)			149089	(LoopStr)	149090	(LoopStr)	LoopStr
468	1782	(Basic)			149091	(Basic)	149092	(Basic)	
			1927	(DataType)	149183	(DataType)	149184	(DataType)	DataType
476	2193	(Basic)			149209	(Basic)	149210	(Basic)	Taint
			2195	(Basic)	149245	(Basic)	149246	(Basic)	
	1879	(Scope)	1880	(Scope)	149149	(Scope)	149150	(Scope)	Scope
	1875	(AddrAlias)	1876	(AddrAlias)	149147	(AddrAlias)	149148	(AddrAlias)	AddrAlias
									Container
	1877	(LContFlow)	2194	(LContFlow)	149243	(LContFlow)	149244	(LContFlow)	LContFlow
									LoopStr
489	1861	(Basic)	1862	(Basic)	149135	(Basic)	149136	(Basic)	

Table D.1: Gap analysis of complexities covered by weakness categories for each test suite.

D.0.2 Observations while reviewing:

In addition to providing the results of the gap analysis, NIST was also provided with an enumeration of additional questions regarding inconsistencies for future consideration with respect to the test suites and the test plan.

- What is the difference between "taint" and "basic"? While I didn't do an exhaustive analysis, most "basic" take a taint from argv.
- What is "Array Index Complexity" and why is it not listed as a complexity? Is it the same as "Array Address Complexity"?
- What is "Alias Index Level" and why is it not listed as a complexity?

- 1845 and 1847, 1846 and 1848, 2146 and 2148, as well as 2145 and 2147, are listed as different complexities ("array index complexity" vs "memory location complexity"), however, their code is exactly the same. What is the reason for this? Is it possible that this inconsistency was the result of a cut-copy-paste that was never caught?
- 1848 should be marked as "Array Index Complexity" for description and complexity (test plan document is incorrect).
- Test case 92 is marked with complexity "Address Alias Level" while "1556" is marked with complexity "Scope". The challenge here is that 92 and 1555 are exactly the same code, 1555 is the pair for 1556, and as such 1556 is a "good" replacement for 92 (bad). They need to be the same complexity don't they? I believe they should both be marked as "Scope".
- I don't think "All" should be listed as the complexity by a specification, since relevant complexities can change over time. Rather the full enumeration of complexities should be provided. See CWE-121 and CWE-122 above. How do you know if they have coverage?
- Has an exhaustive evaluation been conducted to ensure that the complexities for each CWE is complete (See CWE-259 where we have a complexity covered that is not listed as required)? More specifically, for each CWE there may be other complexities that need to be covered that are not included in the specification.
- 2131 is listed as complexity "Basic" when it should be "Container" (in the test plan document).
- Should the "asynchronous" complexity be tested across multiple coding constructs (see CWE-367 for example, where we test two different code complexities)? Presently the test plan covers one permutation of the possible asynchronous enumerations in only one or two coding constructs. This seems insufficient.
- What is the meaning of "Basic"? Should every CWE have a "Basic"? In cases to do with Input Validation, it appears to be a basic taint pass. However, in cases that do not require taint tracing, such as CWE-401, it does not. I believe 1585 should be labeled as "Scope" complexity (as is the pairing 1586).
- Test case 1588 should labeled with complexity "Address Alias Level" (as is its pair).
- Why is "Data Type" not listed as a complexity for CWE-457?
- Why are there no relevant complexities for CWE-489?

Through the gap analysis conducted in Table D.1, it would appear that test suite 45 and 46 were never designed to cover all of the "complexity" classes that were

specified in the accompanying specification. For those complexities that were covered, typically only one variation of the enumeration of possibilities is covered. Additionally, we also noticed the following inconsistencies:

- Is *LoopComp* the same as *LoopStr*?
- Is *Taint* the same as *Basic* (see 1857)?
- Is *Buffer Address Space* the same as *Buffer Address Type* (see 99)?
- The different number of test cases across CWE's in the test suite also provide artificial weighting of results. Should a test suite have the same number of test cases per CWE or have the measurement system for the test suite weighted to treat all weakness classes the same by default.

Finally, while the above analysis helps ensure that the test cases are now paired, and labeled with complexities (row-wise), another pass still needs to be made to ensure that the labeled complexities are actually correct for the code.

Appendix E

Experiment Data

E.1 Dataset NEW: SARD Test Suite 100 and 101(SAMATE Revisions (PR) - FINAL)

Note: file name suffixes for scan version have been removed.

New (Test Case)	File	Vulnerability Categories
/149/045	fmt-bad.c	Format String: 1
/149/047	fmt-bad.c	Format String: 1
/149/049	mem-bad.c	Double Free: 1
/149/051	race-bad.c	
/149/053	tain-bad.c	Command Injection: 1
/149/055	ahdec1-bad.c	
/149/057	ahgets1-bad.c	
/149/059	ahscpy1-bad.c	
/149/061	fmt3-bad.c	Format String: 1
/149/063	fmt5-bad.c	Format String: 1
/149/065	gets1-bad.c	Dangerous Function: 1
/149/067	gets2-bad.c	Buffer Overflow: 1
/149/069	into2-bad.c	Integer Overflow: 1
/149/071	mem1-bad.c	Memory Leak: 1
/149/073	mem2-bad.c	Memory Leak: 1
/149/075	mem3-bad.c	Double Free: 1 Use After Free: 1
/149/077	scpy2-bad.c	Buffer Overflow: Off-by-One: 1
/149/079	scpy7-bad.c	Buffer Overflow: 1
/149/081	scpy8-bad.c	Buffer Overflow: 1
/149/083	scpy9-bad.c	Buffer Overflow: 1
/149/085	heapinspection-bad.c	Memory Leak: Reallocation: 1 Heap Inspection: 1
/149/087	stack_overflow-bad.c	Buffer Overflow: 1

/149/089	uninitialized_variable-bad.c	Uninitialized Variable: 1
/149/091	Unintentional_pointer_scaling-bad.c	
/149/093	xss_basic-bad.c	Cross-Site Scripting: Reflected: 1
/149/095	sql_select-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/097	sql_array-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/099	sql_scope-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/101	race_basic-bad.c	Race Condition: File System Access: 1
/149/103	race_fctptr-bad.c	Race Condition: File System Access: 1
/149/105	Use_of_hardcoded_password1-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/107	dble_free_local_flow-bad.c	Redundant Null Check: 2
/149/109	dble_free_loop-bad.c	Double Free: 1 Use After Free: 1 Memory Leak: 1
/149/111	fmt_string_local_container-bad.c	Format String: 1
/149/113	fmt_string_local_control_flow-bad.c	Format String: 1
/149/115	hardcoded_pass_buffer-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 2 Password Management: Password in Comment: 2 Obsolete: 1
/149/117	hardcoded_pass_container-bad.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/119	hardcoded_pass_local_flow-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/121	hardcoded_pass_loop-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1

/149/123	heap_overflow_array-bad.c	Buffer Overflow: 1
/149/125	heap_overflow_cplx-bad.c	Buffer Overflow: 1 Integer Overflow: 1
/149/127	improper_null_term_basic-bad.c	String Termination Error: 1
/149/129	improper_null_term_basic_@alias-bad.c	String Termination Error: 1
/149/131	improper_null_term_basic_container-bad.c	String Termination Error: 1
/149/133	improper_null_term_basic_taint-bad.c	String Termination Error: 1
/149/135	leftover_debug-bad.c	
/149/137	misused_string_fct-bad.c	Buffer Overflow: 1
/149/139	misused_string_fct_@alias-bad.c	Buffer Overflow: 1
/149/141	misused_string_fct_container-bad.c	Buffer Overflow: 1
/149/143	misused_string_fct_scope-bad.c	Buffer Overflow: 1
/149/145	misused_string_fct_taint-bad.c	Buffer Overflow: 1
/149/147	null_deref_@alias-bad.c	
/149/149	null_deref_scope-bad.c	
/149/151	os_cmd_local_flow-bad.c	Command Injection: 1
/149/153	os_cmd_loop-bad.c	Command Injection: 1
/149/155	os_cmd_scope-bad.c	Command Injection: 1
/149/157	resource_injection_@alias-bad.c	Path Manipulation: 1
/149/159	resource_injection_basic-bad.c	Path Manipulation: 1
/149/161	resource_injection_container-bad.c	Path Manipulation: 1
/149/163	resource_injection_scope-bad.c	Path Manipulation: 1
/149/165	stack_overflow_array_length-bad.c	Buffer Overflow: 1
/149/167	stack_overflow_index_alias-bad.c	Buffer Overflow: Off-by-One: 1
/149/169	stack_overflow_loop-bad.c	Buffer Overflow: Off-by-One: 1
/149/173	xss_@alias-bad.c	Cross-Site Scripting: Reflected: 1
/149/175	xss_container-bad.c	Cross-Site Scripting: Reflected: 1
/149/177	xss_scope-bad.c	Cross-Site Scripting: Reflected: 1
/149/179	memory_leak_container-bad.c	
/149/181	memory_leak_loop-bad.c	Memory Leak: 1
/149/183	unintentional_pointer_scaling_data-bad.c	
/149/185	unchecked_error_condition-bad.c	
/149/187	sql_injection_loop-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/189	memory_leak_basic-bad.c	Memory Leak: 1
/149/191	UninitializedVariable_pointer-bad.c	Uninitialized Variable: 1
/149/193	StackOverflow-bad.c	Buffer Overflow: 4
/149/195	Improper_null_term_BufferAddressType-bad.c	String Termination Error: 1
/149/197	Uninitialized_variable-bad.c	Uninitialized Variable: 1
/149/199	lock_resource-bad.c	
/149/201	HeapOverFlow-bad.c	Buffer Overflow: 1
/149/203	UseAfterFree_container-bad.c	Use After Free: 1
/149/207	UninitializedVariable_DataType-bad.c	Uninitialized Variable: 1
/149/209	NullPointerDereference-bad.c	Null Dereference: 1

/149/215	xss_loop-bad.c	Cross-Site Scripting: Reflected: 1
/149/217	double_free-bad.c	Double Free: 1
/149/219	useafterfree-bad.c	Use After Free: 1
/149/221	use_after_free_scope-bad.c	Use After Free: 1
/149/223	use_after_free_container-bad.c	Use After Free: 1
/149/225	use_after_free_@buffer-bad.c	Use After Free: 1
/149/229	double_free-bad.c	Double Free: 1
/149/231	race_stat_basic-bad.c	Race Condition: File System Access: 1
/149/233	race_stat_fctptr-bad.c	Race Condition: File System Access: 1
/149/237	Format_string_problem-bad.c	Format String: 1
/149/239	use_after_free_@buffer-bad.c	Use After Free: 1
/149/241	os_cmd_injection_basic-bad.c	Command Injection: 1 Integer Overflow: 1 String Termination Error: 1
/149/243	null_deref_local_flow-bad.c	Null Dereference: 1
/149/245	null_deref-bad.c	Null Dereference: 1
/149/247	use_after_free_@alias-bad.c	

Table E.1: SAMATE Replacement for Test Suite 45, SARD 100 Scanned with custom rules 2 (Jan. 7th, 2016).

New (Test Case)	File	Vulnerability Categories
/149/045	fmt-bad.c	Format String: 1
/149/047	fmt-bad.c	Format String: 1
/149/049	mem-bad.c	Double Free: 1
/149/051	race-bad.c	
/149/053	tain-bad.c	Command Injection: 1
/149/055	ahdec1-bad.c	
/149/057	ahgets1-bad.c	
/149/059	ahscpy1-bad.c	
/149/061	fmt3-bad.c	Format String: 1
/149/063	fmt5-bad.c	Format String: 1
/149/065	gets1-bad.c	Dangerous Function: 1
/149/067	gets2-bad.c	Buffer Overflow: 1
/149/069	into2-bad.c	Integer Overflow: 1
/149/071	mem1-bad.c	Memory Leak: 1
/149/073	mem2-bad.c	Memory Leak: 1
/149/075	mem3-bad.c	Double Free: 1 Use After Free: 1
/149/077	scpy2-bad.c	Buffer Overflow: Off-by-One: 1 Dangerous Function: strcpy(): 1
/149/079	scpy7-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/081	scpy8-bad.c	Buffer Overflow: 1
/149/083	scpy9-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/085	heapinspection-bad.c	Memory Leak: Reallocation: 1 Heap Inspection: 1

/149/087	stack_overflow-bad.c	Buffer Overflow: 1
/149/089	uninitialized_variable-bad.c	Uninitialized Variable: 1
/149/091	Unintentional_pointer_scaling-bad.c	
/149/093	xss_basic-bad.c	
/149/095	sql_select-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/097	sql_array-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/099	sql_scope-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/101	race_basic-bad.c	Race Condition: File System Access: 1
/149/103	race_fctptr-bad.c	Race Condition: File System Access: 1
/149/105	Use_of_hardcoded_password1-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/107	dble_free_local_flow-bad.c	Redundant Null Check: 2
/149/109	dble_free_loop-bad.c	Double Free: 1 Use After Free: 1 Memory Leak: 1
/149/111	fmt_string_local_container-bad.c	Format String: 1
/149/113	fmt_string_local_control_flow-bad.c	Format String: 1
/149/115	hardcoded_pass_buffer-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 2 Password Management: Password in Comment: 2 Obsolete: 1
/149/117	hardcoded_pass_container-bad.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/119	hardcoded_pass_local_flow-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1

/149/121	hardcoded_pass_loop-bad.c	Password Management: Empty Password: 1 Password Management: Hardcoded Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/123	heap_overflow_array-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/125	heap_overflow_cplx-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1 Integer Overflow: 1
/149/127	improper_null_term_basic-bad.c	String Termination Error: 1
/149/129	improper_null_term_basic_@alias-bad.c	String Termination Error: 1
/149/131	improper_null_term_basic_container-bad.c	String Termination Error: 1
/149/133	improper_null_term_basic_taint-bad.c	String Termination Error: 1
/149/135	leftover_debug-bad.c	
/149/137	misused_string_fct-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/139	misused_string_fct_@alias-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/141	misused_string_fct_container-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/143	misused_string_fct_scope-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/145	misused_string_fct_taint-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/147	null_deref_@alias-bad.c	
/149/149	null_deref_scope-bad.c	
/149/151	os_cmd_local_flow-bad.c	Dangerous Function: strcpy(): 1 Command Injection: 1
/149/153	os_cmd_loop-bad.c	Dangerous Function: strcpy(): 1 Command Injection: 1
/149/155	os_cmd_scope-bad.c	Dangerous Function: strcpy(): 1 Command Injection: 1
/149/157	resource_injection_@alias-bad.c	Path Manipulation: 1
/149/159	resource_injection_basic-bad.c	Path Manipulation: 1
/149/161	resource_injection_container-bad.c	Path Manipulation: 1
/149/163	resource_injection_scope-bad.c	Path Manipulation: 1
/149/165	stack_overflow_array_length-bad.c	Buffer Overflow: 1
/149/167	stack_overflow_index_alias-bad.c	Buffer Overflow: Off-by-One: 1
/149/169	stack_overflow_loop-bad.c	Buffer Overflow: Off-by-One: 1
/149/173	xss_@alias-bad.c	
/149/175	xss_container-bad.c	
/149/177	xss_scope-bad.c	
/149/179	memory_leak_container-bad.c	
/149/181	memory_leak_loop-bad.c	Memory Leak: 1
/149/183	unintentional_pointer_scaling_data-bad.c	
/149/185	unchecked_error_condition-bad.c	

/149/187	sql_injection_loop-bad.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/189	memory_leak_basic-bad.c	Memory Leak: 1
/149/191	UninitializedVariable_pointer-bad.c	Uninitialized Variable: 1
/149/193	StackOverflow-bad.c	Buffer Overflow: 4
/149/195	Improper_null_term_BufferAddressType-bad.c	String Termination Error: 1
/149/197	Uninitialized_variable-bad.c	Uninitialized Variable: 1
/149/199	lock_resource-bad.c	
/149/201	HeapOverFlow-bad.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/203	UseAfterFree_container-bad.c	Dangerous Function: strcpy(): 1 Use After Free: 1
/149/207	UninitializedVariable_DataType-bad.c	Uninitialized Variable: 1
/149/209	NullPointerDereference-bad.c	Null Dereference: 1
/149/215	xss_loop-bad.c	
/149/217	double_free-bad.c	Double Free: 1
/149/219	useafterfree-bad.c	Use After Free: 1
/149/221	use_after_free_scope-bad.c	Dangerous Function: strcpy(): 1 Use After Free: 1
/149/223	use_after_free_container-bad.c	Dangerous Function: strcpy(): 1 Use After Free: 1
/149/225	use_after_free_@buffer-bad.c	Dangerous Function: strcpy(): 1 Use After Free: 1
/149/229	double_free-bad.c	Dangerous Function: strcpy(): 1 Double Free: 1
/149/231	race_stat_basic-bad.c	Race Condition: File System Access: 1
/149/233	race_stat_fctptr-bad.c	Race Condition: File System Access: 1
/149/237	Format_string_problem-bad.c	Format String: 1
/149/239	use_after_free_@buffer-bad.c	Dangerous Function: strcpy(): 1 Use After Free: 1
/149/241	os_cmd_injection_basic-bad.c	Command Injection: 1 Integer Overflow: 1 String Termination Error: 1
/149/243	null_deref_local_flow-bad.c	Null Dereference: 1
/149/245	null_deref-bad.c	Null Dereference: 1
/149/247	use_after_free_@alias-bad.c	Dangerous Function: strcpy(): 1

Table E.2: SAMATE Replacement for Test Suite 45, SARD 100 Scanned with default rules (Jan. 7th, 2016).

New (Test Case)	File	Vulnerability Categories
/149/046	fmt-good.c	
/149/048	fmt-good.c	Log Forging: 1
/149/050	mem-good.c	

/149/052	race-good.c	
/149/054	tain-good.c	
/149/056	ahdec1-good.c	
/149/058	ahgets1-good.c	
/149/060	ahscpy1-good.c	
/149/062	fmt3-good.c	Log Forging: 1
/149/064	fmt5-good.c	
/149/066	gets1-good.c	
/149/068	gets2-good.c	
/149/070	into2-good.c	Integer Overflow: 1
/149/072	mem1-good.c	
/149/074	mem2-good.c	
/149/076	mem3-good.c	
/149/078	scpy2-good.c	
/149/080	scpy7-good.c	
/149/082	scpy8-good.c	
/149/084	scpy9-good.c	Buffer Overflow: 1
/149/086	heapinspection-good.c	
/149/088	stack_overflow-good.c	
/149/090	uninitialized_variable-good.c	
/149/092	Unintentional_pointer_scaling-good.c	
/149/094	xss_basic-good.c	
/149/096	sql_select-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/098	sql_array-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/100	sql_scope-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/102	race_basic-good.c	
/149/104	race_fctptr-good.c	
/149/106	Use_of_hardcoded_password1-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/108	dble_free_local_flow-good.c	Redundant Null Check: 2
/149/110	dble_free_loop-good.c	Double Free: 1 Use After Free: 1 Memory Leak: 1
/149/112	fmt_string_local_container-good.c	
/149/114	fmt_string_local_control_flow-good.c	

/149/116	hardcoded_pass_buffer-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/118	hardcoded_pass_container-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/120	hardcoded_pass_local_flow-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/122	hardcoded_pass_loop-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/124	heap_overflow_array-good.c	
/149/126	heap_overflow_cplx-good.c	Integer Overflow: 1
/149/128	improper_null_term_basic-good.c	
/149/130	improper_null_term_basic_@alias-good.c	String Termination Error: 1
/149/132	improper_null_term_basic_container-good.c	String Termination Error: 1
/149/134	improper_null_term_basic_taint-good.c	
/149/136	leftover_debug-good.c	Dead Code: 1
/149/138	misused_string_fct-good.c	
/149/140	misused_string_fct_@alias-good.c	
/149/142	misused_string_fct_container-good.c	
/149/144	misused_string_fct_scope-good.c	
/149/146	misused_string_fct_taint-good.c	
/149/148	null_deref_@alias-good.c	
/149/150	null_deref_scope-good.c	
/149/152	os_cmd_local_flow-good.c	Buffer Overflow: 1
/149/154	os_cmd_loop-good.c	
/149/156	os_cmd_scope-good.c	
/149/158	resource_injection_@alias-good.c	String Termination Error: 1
/149/160	resource_injection_basic-good.c	
/149/162	resource_injection_container-good.c	String Termination Error: 1
/149/164	resource_injection_scope-good.c	
/149/166	stack_overflow_array_length-good.c	Buffer Overflow: 1
/149/168	stack_overflow_index_alias-good.c	
/149/170	stack_overflow_loop-good.c	
/149/174	xss_@alias-good.c	
/149/176	xss_container-good.c	
/149/178	xss_scope-good.c	
/149/180	memory_leak_container-good.c	
/149/182	memory_leak_loop-good.c	
/149/184	unintentional_pointer_scaling_data-good.c	
/149/186	unchecked_error_condition-good.c	
/149/188	sql_injection_loop-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/190	memory_leak_basic-good.c	

/149/192	UninitializedVariable_pointer-good.c	
/149/194	StackOverflow-good.c	Buffer Overflow: 5
/149/196	Improper_null_term_BufferAddressType-good.c	String Termination Error: 1
/149/198	Uninitialized_variable-good.c	
/149/200	lock_resource-good.c	
/149/202	HeapOverFlow-good.c	
/149/204	UseAfterFree_container-good.c	
/149/208	UninitializedVariable_DataType-good.c	Dead Code: 1
/149/210	NullPointerDereference-good.c	Null Dereference: 1
/149/216	xss_loop-good.c	
/149/218	double_free-good.c	
/149/220	useafterfree-good.c	Null Dereference: 1
/149/222	use_after_free_scope-good.c	
/149/224	use_after_free_container-good.c	
/149/226	use_after_free_@buffer-good.c	
/149/230	double_free-good.c	
/149/232	race_stat_basic-good.c	
/149/234	race_stat_fctptr-good.c	
/149/238	Format_string_problem-good.c	
/149/240	use_after_free_@buffer-good.c	
/149/242	os_cmd_injection_basic-good.c	Command Injection: 1 Integer Overflow: 2 String Termination Error: 2
/149/244	null_deref_local_flow-good.c	Dead Code: 1
/149/246	null_deref-good.c	Null Dereference: 1
/149/248	use_after_free_@alias-good.c	

Table E.3: SAMATE Replacement for Test Suite 46, SARD 101 Scanned with custom rules (Jan. 7th, 2016).

New (Test Case)	File	Vulnerability Categories
/149/046	fmt-good.c	
/149/048	fmt-good.c	Log Forging: 1
/149/050	mem-good.c	
/149/052	race-good.c	
/149/054	tain-good.c	Command Injection: 1
/149/056	ahdec1-good.c	
/149/058	ahgets1-good.c	
/149/060	ahscpy1-good.c	
/149/062	fmt3-good.c	Log Forging: 1
/149/064	fmt5-good.c	
/149/066	gets1-good.c	
/149/068	gets2-good.c	
/149/070	into2-good.c	Integer Overflow: 1
/149/072	mem1-good.c	
/149/074	mem2-good.c	
/149/076	mem3-good.c	
/149/078	scpy2-good.c	Dangerous Function: strcpy(): 1

/149/080	scpy7-good.c	Dangerous Function: strcpy(): 1
/149/082	scpy8-good.c	
/149/084	scpy9-good.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1
/149/086	heapinspection-good.c	Dangerous Function: strcpy(): 1
/149/088	stack_overflow-good.c	
/149/090	uninitialized_variable-good.c	
/149/092	Unintentional_pointer_scaling-good.c	
/149/094	xss_basic-good.c	
/149/096	sql_select-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/098	sql_array-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/100	sql_scope-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 SQL Injection: 1 Password Management: Password in Comment: 2 Integer Overflow: 1 Obsolete: 1
/149/102	race_basic-good.c	
/149/104	race_fctptr-good.c	
/149/106	Use_of_hardcoded_password1-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/108	dble_free_local_flow-good.c	Redundant Null Check: 2
/149/110	dble_free_loop-good.c	Double Free: 1 Use After Free: 1 Memory Leak: 1
/149/112	fmt_string_local_container-good.c	
/149/114	fmt_string_local_control_flow-good.c	
/149/116	hardcoded_pass_buffer-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/118	hardcoded_pass_container-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/120	hardcoded_pass_local_flow-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1

/149/122	hardcoded_pass_loop-good.c	Password Management: Empty Password: 1 Password Management: Password in Comment: 2 Obsolete: 1
/149/124	heap_overflow_array-good.c	
/149/126	heap_overflow_cplx-good.c	Integer Overflow: 1
/149/128	improper_null_term_basic-good.c	
/149/130	improper_null_term_basic_@alias-good.c	String Termination Error: 1
/149/132	improper_null_term_basic_container-good.c	String Termination Error: 1
/149/134	improper_null_term_basic_taint-good.c	
/149/136	leftover_debug-good.c	Dead Code: 1
/149/138	misused_string_fct-good.c	
/149/140	misused_string_fct_@alias-good.c	
/149/142	misused_string_fct_container-good.c	
/149/144	misused_string_fct_scope-good.c	
/149/146	misused_string_fct_taint-good.c	
/149/148	null_deref_@alias-good.c	
/149/150	null_deref_scope-good.c	
/149/152	os_cmd_local_flow-good.c	Dangerous Function: strcpy(): 1 Buffer Overflow: 1 Command Injection: 1
/149/154	os_cmd_loop-good.c	Dangerous Function: strcpy(): 1 Command Injection: 1
/149/156	os_cmd_scope-good.c	Dangerous Function: strcpy(): 1 Command Injection: 1
/149/158	resource_injection_@alias-good.c	Path Manipulation: 1 String Termination Error: 1
/149/160	resource_injection_basic-good.c	Path Manipulation: 1
/149/162	resource_injection_container-good.c	Path Manipulation: 1 String Termination Error: 1
/149/164	resource_injection_scope-good.c	Path Manipulation: 1
/149/166	stack_overflow_array_length-good.c	Buffer Overflow: 1
/149/168	stack_overflow_index_alias-good.c	
/149/170	stack_overflow_loop-good.c	
/149/174	xss_@alias-good.c	
/149/176	xss_container-good.c	
/149/178	xss_scope-good.c	
/149/180	memory_leak_container-good.c	
/149/182	memory_leak_loop-good.c	
/149/184	unintentional_pointer_scaling_data-good.c	
/149/186	unchecked_error_condition-good.c	
/149/188	sql_injection_loop-good.c	Password Management: Empty Password: 1 Password Management: 2 Unreleased Resource: 1 Password Management: Password in Comment: 2 SQL Injection: 1 Integer Overflow: 1 Obsolete: 1
/149/190	memory_leak_basic-good.c	
/149/192	UninitializedVariable_pointer-good.c	
/149/194	StackOverflow-good.c	Buffer Overflow: 5

/149/196	Improper_null_term_BufferAddressType-good.c	String Termination Error: 1
/149/198	Uninitialized_variable-good.c	
/149/200	lock_resource-good.c	
/149/202	HeapOverFlow-good.c	Dangerous Function: strcpy(): 1
/149/204	UseAfterFree_container-good.c	Dangerous Function: strcpy(): 1
/149/208	UninitializedVariable_DataType-good.c	Dead Code: 1
/149/210	NullPointerDereference-good.c	Null Dereference: 1
/149/216	xss_loop-good.c	
/149/218	double_free-good.c	
/149/220	useafterfree-good.c	Null Dereference: 1
/149/222	use_after_free_scope-good.c	Dangerous Function: strcpy(): 1
/149/224	use_after_free_container-good.c	Dangerous Function: strcpy(): 1
/149/226	use_after_free_@buffer-good.c	Dangerous Function: strcpy(): 1
/149/230	double_free-good.c	Dangerous Function: strcpy(): 1
/149/232	race_stat_basic-good.c	
/149/234	race_stat_fctptr-good.c	
/149/238	Format_string_problem-good.c	
/149/240	use_after_free_@buffer-good.c	Dangerous Function: strcpy(): 1
/149/242	os_cmd_injection_basic-good.c	Command Injection: 1 Integer Overflow: 2 String Termination Error: 2
/149/244	null_deref_local_flow-good.c	Dead Code: 1
/149/246	null_deref-good.c	Null Dereference: 1
/149/248	use_after_free_@alias-good.c	Dangerous Function: strcpy(): 1

Table E.4: SAMATE Replacement for Test Suite 46, SARD 101 Scanned with default rules (Jan. 7th, 2016).

Appendix F

Custom Rules

```
<?xml version="1.0" encoding="UTF-8"?>
<RulePack xmlns="xmlns://www.fortifysoftware.com/schema/rules"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="RulePack">
    <RulePackID>C0BE39AA-1E47-4C07-A963-A5A8FCB3716A</RulePackID>
    <SKU>SKU-C0BE39AA-1E47-4C07-A963-A5A8FCB3716A</SKU>
    <Name><![CDATA[cb_samf.xml]]></Name>
    <Version>1.0</Version>
    <Description><![CDATA[Description for cb_samf.xml]]></Description>
    <Rules version="3.16">
        <RuleDefinitions>

            <DataflowSourceRule formatVersion="3.2" language="cpp">
                <RuleID>15901788-53F2-4C7D-A3B4-AB914F77CDC9</RuleID>
                <TaintFlags>+XSS,+WEB</TaintFlags>
                <FunctionIdentifier>
                    <NamespaceName>
                        <Pattern/>
                    </NamespaceName>
                    <ClassName>
                        <Pattern/>
                    </ClassName>
                    <FunctionName>
                        <Pattern>cgiFormString|cgiFormStringNoNewlines</Pattern>
                    </FunctionName>
                    <ApplyTo implements="true" overrides="true" extends="true"/>
                </FunctionIdentifier>
                <OutArguments>1</OutArguments>
            </DataflowSourceRule>
```

```
29
30          <DataflowSinkRule formatVersion="3.16" language="cpp">
31          <MetaInfo>
32              <Group name="altcategoryCWE">CWE ID 79, 80</Group>
33          </MetaInfo>
34          <RuleID>FA43523C-4858-4A7A-8914-2217C4B6C7FC0</RuleID>
35          <VulnKingdom>Input Validation and Representation</VulnKingdom>
36          <VulnCategory>Cross-Site Scripting</VulnCategory>
37          <VulnSubcategory>Reflected</VulnSubcategory>
38          <DefaultSeverity>4.0</DefaultSeverity>
39          <Description ref="desc.dataflow.cpp. cross_site_scripting_reflected ">
40              <Explanation append="true"><![CDATA[This issue is being reported by a
                    custom rule.]]></Explanation>
41          </Description>
42          <Sink>
43              <InArguments>2...</InArguments>
44              <Conditional>
45                  <And>
46                      <TaintFlagSet taintFlag="XSS"/>
47                      <Or>
48                          <TaintFlagSet taintFlag="WEB"/>
49                          <TaintFlagSet taintFlag="GUI_FORM"/>
50                          <TaintFlagSet taintFlag="FORM"/>
51                          <TaintFlagSet taintFlag="NETWORK"/>
52                      </Or>
53                      <Not>
54                          <TaintFlagSet taintFlag=
55                          "VALIDATED_CROSS_SITE_SCRIPTING_REFLECTED"/>
56                      </Not>
57                  </And>
58              </Conditional>
59          </Sink>
60          <FunctionIdentifier>
61              <NamespaceName>
62                  <Pattern/>
63              </NamespaceName>
64              <ClassName>
65                  <Pattern/>
66              </ClassName>
67              <FunctionName>
68                  <Pattern>fprintf</Pattern>
69              </FunctionName>
```

```
70              <ApplyTo implements="true" overrides="true" extends="true"/>
71          </FunctionIdentifier>
72          </DataflowSinkRule>
73
74          <DataflowSinkRule formatVersion="3.16" language="cpp">
75          <MetaInfo>
76              <Group name="altcategoryCWE">CWE ID 79, 80</Group>
77          </MetaInfo>
78          <RuleID>FA43523C-4858-4A7A-8914-2217C4B6C7FC1</RuleID>
79          <VulnKingdom>Input Validation and Representation</VulnKingdom>
80          <VulnCategory>Cross-Site Scripting</VulnCategory>
81          <VulnSubcategory>Persistent</VulnSubcategory>
82          <DefaultSeverity>4.0</DefaultSeverity>
83          <Description ref="desc.dataflow.cpp. cross_site_scripting_persistent ">
84              <Explanation append="true"><![CDATA[This issue is being reported by a
                    custom rule.]]></Explanation>
85          </Description>
86          <Sink>
87              <InArguments>2...</InArguments>
88              <Conditional>
89                  <And>
90                      <TaintFlagSet taintFlag="XSS"/>
91                      <Or>
92                          <TaintFlagSet taintFlag="DATABASE"/>
93                          <TaintFlagSet taintFlag="LDAP"/>
94                          <TaintFlagSet taintFlag="NAMING"/>
95                          <TaintFlagSet taintFlag="XML"/>
96                          <TaintFlagSet taintFlag="WEBSERVICE"/>
97                      </Or>
98                      <Not>
99                          <TaintFlagSet taintFlag=
100                         "VALIDATED_CROSS_SITE_SCRIPTING_PERSISTENT"/>
101                     </Not>
102                 </And>
103             </Conditional>
104         </Sink>
105         <FunctionIdentifier>
106             <NamespaceName>
107                 <Pattern/>
108             </NamespaceName>
109             <ClassName>
110               <Pattern/>
```

```
111             </ClassName>
112             <FunctionName>
113                 <Pattern>fprintf</Pattern>
114             </FunctionName>
115             <ApplyTo implements="true" overrides="true" extends="true"/>
116         </FunctionIdentifier>
117         </DataflowSinkRule>
118
119         <!-- Removal of noisy semantic rules -->
120         <!-- Suppress the strcpy semantic rule -->
121         <SuppressionRule formatVersion="6.10">
122             <RuleID>BA40B07C-72BC-4732-B24C-9E9C7CFF1089</RuleID>
123         </SuppressionRule>
124
125         <!-- Cleanup of SQL Injection Results that use a combined approach -->
126         <!-- Taint to be added when strings are properly escaped (Validated)-->
127         <DataflowCleanseRule formatVersion="6.10" language="cpp">
128             <RuleID>ABECDC8A-0BA6-4BDA-A4D2-E713DBD43E9D</RuleID>
129             <TaintFlags>+VALIDATED_SQL_INJECTION</TaintFlags>
130             <FunctionIdentifier>
131                 <NamespaceName>
132                     <Pattern/>
133                 </NamespaceName>
134                 <ClassName>
135                     <Pattern/>
136                 </ClassName>
137                 <FunctionName>
138                     <Pattern>mysql_real_escape_string</Pattern>
139                 </FunctionName>
140                 <ApplyTo implements="true" overrides="true" extends="true"/>
141             </FunctionIdentifier>
142             <OutArguments>1</OutArguments>
143         </DataflowCleanseRule>
144
145         <!-- Suppress semantic SQLi rule (normally hidden by DF rules) -->
146         <SuppressionRule formatVersion="6.10">
147             <RuleID>0C48C455-7B55-491C-9169-10CFA8B818B0</RuleID>
148         </SuppressionRule>
149
150         <!-- Dataflow passthrough for allowed() function in path manipulation/resource
                injection -->
151         <DataflowPassthroughRule formatVersion="6.10" language="cpp">
```

```
152             <RuleID>9DF57308-02AF-4027-B1E0-F6F74DBA20DE</RuleID>
153             <TaintFlags>+VALIDATED_PATH_MANIPULATION</TaintFlags>
154             <FunctionIdentifier>
155                 <NamespaceName>
156                     <Pattern/>
157                 </NamespaceName>
158                 <ClassName>
159                     <Pattern/>
160                 </ClassName>
161                 <FunctionName>
162                     <Pattern>allowed</Pattern>
163                 </FunctionName>
164                 <ApplyTo implements="true" overrides="true" extends="true"/>
165             </FunctionIdentifier>
166             <InArguments>0</InArguments>
167             <OutArguments>0</OutArguments>
168         </DataflowPassthroughRule>
169
170         <CharacterizationRule formatVersion="6.10" language="cpp">
171             <RuleID>474968BB-094B-4553-8003-605C0913A2A6</RuleID>
172             <Definition><![CDATA[
173                 strcpy(dest, src) {
174                 BufferWrite(dest, 0, src.$strlen + 1)
175                 Ensures(dest.$strlen' == src.$strlen)
176                 Ensures(dest.$offset' == dest.$offset)
177                 Ensures(dest.$size' == dest.$size)
178                 Ensures(return.$strlen == src.$strlen)
179                 Ensures(return.$offset == dest.$offset)
180                 Ensures(return.$size == dest.$size)
181                 controlflow:
182                 Copy(in=dest, out=return)
183                 }
184             ]]></Definition>
185         </CharacterizationRule>
186
187         <!-- Dataflow passthrough rule for purify(pointer) does not work for 1 case due
                to SCA limitation -->
188         <!-- Mark check as a validated command injection passthrough (removes critical
                'command injection' dataflow) -->
189         <DataflowPassthroughRule formatVersion="6.10" language="cpp">
190             <RuleID>54DC1FF0-65B4-46A8-9C01-0B71BAA94F2E</RuleID>
191             <TaintFlags>+VALIDATED_COMMAND_INJECTION</TaintFlags>
```

```
192             <FunctionIdentifier>
193                 <NamespaceName>
194                     <Pattern/>
195                 </NamespaceName>
196                 <ClassName>
197                     <Pattern/>
198                 </ClassName>
199                 <FunctionName>
200                     <Pattern>check</Pattern>
201                 </FunctionName>
202                 <ApplyTo implements="true" overrides="true" extends="true"/>
203             </FunctionIdentifier>
204             <InArguments>0</InArguments>
205             <OutArguments>0</OutArguments>
206         </DataflowPassthroughRule>
207
208         <!-- Mark purify as a validated command injection passthrough (removes critical
                'command injection' dataflow) -->
209         <DataflowPassthroughRule formatVersion="6.10" language="cpp">
210             <RuleID>171E43E4-FD7B-47EA-80FC-1333459F555F</RuleID>
211             <TaintFlags>+VALIDATED_COMMAND_INJECTION</TaintFlags>
212             <FunctionIdentifier>
213                 <NamespaceName>
214                 <Pattern/>
215                     </NamespaceName>
216                 <ClassName>
217                     <Pattern/>
218                 </ClassName>
219                 <FunctionName>
220                     <Pattern>purify</Pattern>
221                 </FunctionName>
222                 <ApplyTo implements="true" overrides="true" extends="true"/>
223             </FunctionIdentifier>
224             <InArguments>0</InArguments>
225             <OutArguments>0</OutArguments>
226         </DataflowPassthroughRule>
227
228         <!-- Removal of noisy semantic rules -->
229         <!-- Suppress the system semantic rule (removes low 'command injection' -->
230         <!-- Note: it would appear that there is a duplicate rule -->
231         <SuppressionRule formatVersion="6.10">
232             <RuleID>D9BA784F-6843-4D49-8A07-B8095678E25B</RuleID>
```

```
233             </SuppressionRule>
234             <SuppressionRule formatVersion="6.10">
235                 <RuleID>8DAF9A65-6849-47C7-8BED-F657DE7D142C</RuleID>
236             </SuppressionRule>
237         </RuleDefinitions>
238     </Rules>
239 </RulePack>
```