4-1-2016

A Quantified Model of Security Policies, with an Application for Injection-Attack Prevention

Donald James Ray

A Quantified Model of Security Policies, with an Application for Injection-Attack Prevention

by

Donald Ray

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Date of Approval:
March 25, 2016

Keywords: Code injection, Security metrics, Gray policies

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation generalizes traditional models of security policies, from specifications of *whether* programs are secure, to specifications of *how* secure programs are. This is a generalization from qualitative, black-and-white policies to quantitative, gray policies. Included are generalizations from traditional definitions of safety and liveness policies to definitions of gray-safety and gray-liveness policies. These generalizations preserve key properties of safety and liveness, including that the intersection of safety and liveness is a unique allow-all policy and that every policy can be written as the conjunction of a single safety and a single liveness policy. It is argued that the generalization provides several benefits, including that it serves as a unifying framework for disparate approaches to security metrics, and that it separates—in a practically useful way—specifications of how secure systems are from specifications of how secure users require their systems to be. To demonstrate the usefulness of the new model, policies for mitigating injection attacks (including both code- and noncode-injection attacks) are explored. These policies are based on novel techniques for detecting injection attacks that avoid many of the problems associated with existing mechanisms for preventing injection attacks.

CHAPTER 1

INTRODUCTION

Computer-security policies have traditionally been modeled as predicates, partitioning the secure from the insecure system behaviors. Policies partition behaviors by specifying constraints like "only administrators may write to files", "packets destined for port 120 must be logged", or "all array accesses must be bounds-checked". These are qualitative, black-and-white constraints that can be used to decide whether a given system is secure.

This dissertation generalizes the qualitative, black-and-white model of policies to a quantitative, gray model. Instead of specifying *whether* systems are secure, gray policies specify *how* secure systems are. For example, a gray policy for array-bounds checking might consider that checking every array access makes a program 100% secure and that each unchecked access decimates a program's current rating.

Gray policies are useful because users are often unwilling to pay the costs required to achieve 100% security according to some policy. As is well understood, enforcement costs can be high, typically in the form of:

1. performance overhead (e.g., due to increased runtime checks),
2. code-size overhead (e.g., due to inlined monitoring code),
3. decreased usability (e.g., due to authentication procedures), and
4. consumption of other system resources (e.g., due to security checks draining batteries or security logs draining file-system space).

To make an analogy with the physical world, safes are not rated as secure or insecure, but rather by the estimated amount of time needed to penetrate them with a given set of tools. Such a quantitative rating enables consumers to weigh the security metric against other metrics, such as size, weight, price, and availability, when choosing a safe to buy. Importantly, a choice made in one context may differ from a choice made in another context, depending on the priorities of the purchaser and resources available.

In this dissertation's framework, a gray policy specifies a system's security rating, while a *silhouette judge* specifies a user's security requirements. Returning to the safe analogy, a silhouette judge is like a consumer's purchasing-decision algorithm that inputs a safe's security rating and, combining it with the safe's other attributes, outputs a buy or don't-buy decision.

Thus, this dissertation's framework separates the intuitively distinct specifications of how secure systems are (gray policies) from how secure users require their systems to be (silhouette judges). This separation enables users with different security requirements to use the same gray policy in different ways, by specifying different silhouette judges. For example, in the context of high-performance systems research users might require 0% security (e.g., no array-bounds checking), while in the context of flight-navigation software users might require 100% security.

There are additional benefits of the gray model over the black-and-white model. Gray policies enable users to compare the security of different systems when choosing which to use. In the black-and-white model, a user who can't afford to run a "secure" web browser has to choose between other browsers only known to be "insecure"; in the gray model, the same user could choose the most secure of the affordable browsers. As another potential benefit, consumers often base purchasing decisions on measurable attributes, so quantifying security could drive demand for security improvements, even ones that degrade performance by 10–20% or more, thus countering the arguments of some developers that such security overheads are intolerable [48].

1.1 Roadmap

This dissertation combines and extends ideas first presented in earlier research papers [94, 95, 96]. It concerns the previously unrelated fields of formal security policies, quantified security metrics, and code-injection attacks, and is organized in the following manner:

1. Chapter 2 presents a new, quantified model of formal security policies. This model generalizes existing definitions of policies, properties, safety, liveness, hypersafety, and hyperliveness. It is shown that the new model is indeed a generalization, in that every black-and-white policy is also a gray policy, every black-and-white safety property is also a gray safety property, etc.

2. Chapter 3 shows how this dissertation's model of gray policies can serve as a unifying framework for many disparate approaches to security metrics, and how gray policies might be constructed from various security metrics, as well as existing black-and-white policies.

3. Chapter 4 delves into the problem of injection attacks, a pervasive threat to software security. Problems with existing techniques for preventing injection attacks are investigated, and new techniques which avoid these problems are presented. The novel techniques are then used to construct gray policies and silhouette judges to mitigate and prevent injection attacks.

4. Chapter 5 concludes.

In lieu of having a single related work section to cover all of these subtopics, the related works for each topic are discussed in their respective chapters.

CHAPTER 2

FROM BLACK-AND-WHITE TO GRAY POLICIES

The idea to quantify security is not new (e.g., [25, 75, 4, 31, 54, 46, 78]).

However, the extensive research into general-purpose models of policies has considered them to be predicates and therefore black and white (e.g., [101, 43, 64, 35, 37, 30]). Many interesting results have come from these qualitative models of policies, including definitions of safety and liveness properties, which are tied to particular classes of enforcement mechanisms, and proofs that every black-and-white property is the conjunction of one safety property and one liveness property.

This chapter[1] contributes a more general, quantitative model of policies and properties, and demonstrates that the existing results of the black-and-white model carry through into the gray model.

2.1 Gray Policies

Policies reason about systems, which execute *events*. Let E be a non-empty, countable set of events, with metavariable e ranging over individual events. Intuitively, E is the system API and might contain instructions for manipulating system resources.

A system *trace*, or *execution*, x, is a possibly infinite sequence of pairs of events called *exchanges*. The events in an exchange $\langle e, e' \rangle$ indicate

1. an event e the system attempts to execute, and

2. an event e' that actually executes.

[1]The ideas and much of the text in this chapter first appeared in [96]. Permission to use this material is in Appendix A.

For example, the trace:

$$\langle sti(0, 0x9ABC), sti(0, 0x1ABC)\rangle \, \langle rdr(4, 0x8FFF), rdr(4, 0x0FFF)\rangle$$

indicates that the *target system* being reasoned about, for example an application program, attempted to store the immediate value 0 at memory address 0x9ABC, but 0 was instead written at address 0x1ABC, due to the involvement of a runtime mechanism such as a virtual-memory manager. The second exchange in the trace also shows involvement of a runtime mechanism, again decreasing the memory address being accessed by 2^{15}.

This model of traces as sequences of exchanges is general, in part because it clarifies the effects of runtime mechanisms; such clarification improves expressiveness [65, 37]. In cases where policies require no runtime support, such as statically enforced policies, the first and second events in exchanges will be the same.

Some additional notation will be useful. A set of events E determines the set of possible exchanges $\mathcal{E}$. Given exchange set $\mathcal{E}$, $\mathcal{E}^*$ denotes the set of all finite executions (i.e., finite sequences of exchanges), $\mathcal{E}^\omega$ denotes the set of all infinite executions, and $\mathcal{E}^\infty$ denotes the set of all finite and infinite executions. Also, $x \preceq y$ and $y \succeq x$ mean that execution $x \in \mathcal{E}^*$ is a *prefix* of execution $y \in \mathcal{E}^\infty$. Finally, shorthand quantifications will be used in formulae; for example, $\exists x \preceq y : F$ means $\exists x \in \mathcal{E}^* : (x \preceq y \wedge F)$, while $\forall x \succeq y : F$ means $\forall x \in \mathcal{E}^\infty : (x \succeq y \Rightarrow F)$.

2.1.1 Policies and Properties

The black-and-white model defines *policies* P as predicates over target systems; the policy returns a yes-no response to a given target system, to indicate whether that system is secure [101]. A target system X is modeled as the set of executions it can produce, for example, all possible runs of an application program. Hence, on a system with exchange set $\mathcal{E}$, X is a subset of $\mathcal{E}^\infty$, or, equivalently, a predicate over target systems.

Definition 1 ([101])**.** *A* black-and-white policy *is a* $P : 2^{\mathcal{E}^\infty} \rightarrow \{false, true\}$.

The gray model defines policies G as functions mapping target systems not to false/true values, but to a real number between 0 and 1, with greater numbers indicating higher security. Gray policies generalize black-and-white policies because false/true values in the black-and-white model can always be encoded as 0/1 values in the gray model.

Definition 2. *A* gray policy *is a* $G : 2^{\mathcal{E}^\infty} \rightarrow \mathbb{R}_{[0,1]}$.

In the black-and-white model, *properties* are policies that place no constraints on the relationships between executions [101]. It can be determined whether a target system satisfies a property by examining each possible trace in isolation; if every trace is valid in isolation (according to some predicate p over individual traces), then the policy as a whole is satisfied.

Definition 3 ([101])**.** *A policy P is a* black-and-white property *iff* $\exists \left(p : \mathcal{E}^\infty \rightarrow \{false, true\}\right) : \forall X \subseteq \mathcal{E}^\infty : P\left(X\right) = \left(\forall x \in X : p\left(x\right)\right)$.

The gray model also considers a policy to be a property when the policy's value for a given a set of executions can be determined by examining each execution in isolation. While the black-and-white model determines the policy's value $P(X)$ as the *conjunction* of the values of $p(x)$, for all $x \in X$, the gray model determines the policy's value $G(X)$ as the *infimum* (inf) of the values of $g(x)$, for all $x \in X$. Here g, like p, is a function over individual traces.

Definition 4. *A policy G is a* gray property *iff* $\exists \left(g : \mathcal{E}^\infty \rightarrow \mathbb{R}_{[0,1]}\right) : \forall X \subseteq \mathcal{E}^\infty : G\left(X\right) = \inf\{g\left(x\right) \mid x \in X\}$.

Gray properties generalize black-and-white properties because the conjunction of a set of false/true values always equals the infimum of a set of corresponding 0/1 values. Unfortunately, the use of the infimum precludes limiting the range of security values in the gray model to computable reals; computable reals are not closed under infimum operations [106].

It is often convenient to identify a property by the individual-trace function (p or g) it uses. The following propositions show that there is no ambiguity in doing so, due to the one-to-one correspondence between a p or g function and the property it induces.

Proposition 1. *There is a one-to-one correspondence between a black-and-white property P and its trace-predicate p.*

Proof. First, it is proved that every trace-predicate corresponds to exactly one black-and-white property. Assume there exists a trace predicate p such that p corresponds to two different black-and-white properties P and P'. By the definition of a property, $P(X) = (\forall x \in X : p(x)) = P'(X)$, so for all X, $P(X) = P'(X)$. Thus P and P' are the same property, a contradiction.

Next, it is proved that no black-and-white property has more than one trace predicate. Assume there exists a black-and-white property P and two different trace predicates p and p' such that for all $X \subseteq \mathcal{E}^\infty$, $P(X) = (\forall x \in X : p(x)) = (\forall x \in X : p'(x))$. Consider the set of all target systems that only produce a single execution $\{\{x\} \mid x \in \mathcal{E}^\infty\}$. For each target system T in this set, $P(T)$ holds if and only if $\forall t \in T : p(t)$ and $\forall t \in T : p'(t)$. Since each target system T only contains a single execution x, $p(x) = p'(x)$ for all traces. Thus p and p' are the same trace predicate, a contradiction. $\square$

Proposition 2. *There is a one-to-one correspondence between a gray property G and its gray function g.*

Proof. First, it is proved that every gray function induces exactly one gray property. Assume there exists a gray function g such that g induces two different gray properties G and G'. By the definition of a gray property, $G(X) = \inf\{g(x) \mid x \in X\} = \mathcal{G}'(X)$, so for all X, $G(X) = \mathcal{G}'(X)$. Thus G and G' are the same property, a contradiction.

Next, it is proved that no gray property has more than one gray function g. Assume that there exists a gray property G and two different gray functions g and g' such that for all $X \subseteq \mathcal{E}^\infty$, $G(X) = \inf\{g(x) \mid x \in X\} = \inf\{g'(x) \mid x \in X\}$. Consider the set of all targets that only produce a single execution $\{\{x\} \mid x \in \mathcal{E}^\infty\}$. For each target T in this set, $G(T) = \inf\{g(t) \mid t \in T\} = \inf\{g(t) \mid t \in T\}$. But each set T only contains a single execution t, so $G(T) = g(t) = g'(t)$ for all traces. Thus, g and g' are the same function, a contradiction. $\square$

2.1.2 Safety and Liveness

Two subsets of black-and-white properties have been studied extensively: safety and liveness properties [61, 1]. These sets are fundamentally intertwined with the sets of properties that can be enforced in practice [101, 64, 37, 2, 38].

2.1.2.1 Safety

Black-and-white safety properties partition "secure" from "insecure" traces in such a way that every insecure trace has a finite insecure prefix that can never become secure [61].

Definition 5 ([61])**.** *A property p is* black-and-white safety *iff* $\forall x \in \mathcal{E}^\infty : (\neg p(x) \Rightarrow \exists x' \preceq x : \forall y \succeq x' : \neg p(y))$.

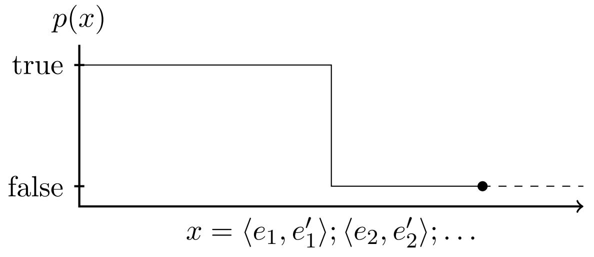
An equivalent, perhaps more intuitive, definition of black-and-white safety is the set of properties that are both prefix- and omega-closed [37]. Prefix-closed means that all prefixes of secure traces are secure, while omega-closed means the converse, that if all prefixes of a trace x are secure then so must be x.

Definition 6 (Equivalent definition of safety [37])**.** *A property p is* black-and-white safety *iff* $\forall x \in \mathcal{E}^\infty : p(x) = (\forall x' \preceq x : p(x'))$.

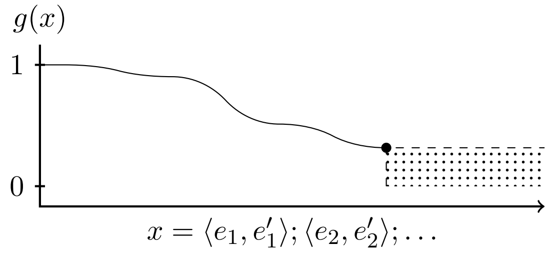
This formalization of black-and-white safety has an interesting similarity to the formalization of black-and-white properties (Definition 3; in both cases, an entity is secure exactly when all of its "simpler parts" are secure.

It can be seen from these definitions that black-and-white safety properties require traces to be as secure as their least-secure prefix; security cannot increase as traces proceed. Figure 2.1(a) plots the general shape of a trace's security as considered by a black-and-white safety property.

Gray safety properties also require traces to have nonincreasing security, as shown in Figure 2.1(b). However, with gray safety, the requirement that traces be as secure as their least-secure prefix must be modified—infinite traces may not have a least-secure prefix. To handle such cases the *infimum* is again used.



(a) Black-and-white Safety



(b) Gray Safety

Figure 2.1. The security of traces as they proceed according to safety properties. The security level is according to (a) a black-and-white safety property and (b) a gray safety property. The dotted lines and shaded area represent the possible security values of the executions' extensions. In all cases, security levels are nonincreasing.

Definition 7. *Formally, property g is* gray safety *iff* $\forall x \in \mathcal{E}^\infty : g(x) = \inf\{g(x') \mid x' \preceq x\}$.

This formalization of gray safety retains the similarity, present in the black-and-white model, between the definitions of properties and safety.

As an example, the gray property described earlier, specifying that a trace's security level gets decimated with each unchecked array access, is a gray safety property. Traces according to this policy begin as 100% secure (before any exchanges occur) and can only proceed to lower security. In the limit, a trace containing an infinite number of unchecked array accesses has 0% security, because the infimum of $\{1, 0.9, 0.81, 0.729, \ldots\}$ is 0.

Because Definitions 5 and 6 are equivalent, and Definition 6 could be "grayified" into Definition 7, it is tempting to also "grayify" Definition 5 into the following:

$$\forall x \in \mathcal{E}^\infty : g(x) < 1 \Rightarrow \exists x' \preceq x : \forall y \succeq x' : g(y) \leq g(x).$$

Unfortunately, this definition is not equivalent to Definition 7 because it excludes gray properties that should be safety. For example, a gray property that halves the security value of an execution for every exchange it includes should be gray safety (and is, according to Definition 7). However, this property maps infinite executions to 0, and all finite prefixes of those infinite executions to values greater than 0. Since all prefixes have a higher security than their infinite extensions, none are suitable choices for x' in the definition above.

To fix this problem, the definition could be relaxed so that extensions of x' need to have a security value that is no higher than the security value of x', rather than the security value of x:

$$\forall x \in \mathcal{E}^\infty : g(x) < 1 \Rightarrow \exists x' \preceq x : \forall y \succeq x' : g(y) \leq g(x').$$

This definition is not equivalent to Definition 7 either; it does not require reductions in security to occur at some finite point. For example, a property could map all finite executions to 1 and all infinite executions to 0. In the black-and-white world, such termination properties are not safety properties; it would be incorrect to consider their gray versions to be safety.

Although Definitions 5 and 6 are equivalent, the fact that Definition 6 can be generalized into the gray model and Definitions 5 cannot suggests that Definition 6 is a better definition of safety, at least in the sense of being more amenable to generalization.

The following proposition shows that gray safety properties are proper generalizations of black-and-white safety properties. In other words, if a black-and-white safety property were to be expressed in the gray world, then it would be a gray safety property, and if a black-and-white property were to be translated into a gray safety property, then it must have been a black-and-white safety property.

Proposition 3. *Let P be a black-and-white property with predicate p. Define a gray function g as follows:*

$$g(x) = \begin{cases} 1, & \text{if } p(x) \\ 0, & \text{if } \neg p(x) \end{cases}$$

Then, g induces a gray safety property iff P is a black-and-white safety property.

Proof. First, it is proved that if P is a black-and-white safety property, then g induces a gray safety property. For an arbitrary execution x, either $p(x)$ or $\neg p(x)$. In either case, it must be shown that $g(x) = \inf\{g(x') \mid x' \preceq x\}$. If $p(x)$, then by Definition 6, $\forall x' \preceq x, p(x')$. By the construction of g, $\forall x' \preceq x, g(x') = 1$ and thus $\inf\{g(x') \mid x' \preceq x\} = 1 = g(x)$. If $\neg p(x)$, there must be some $x' : x' \preceq x \wedge \neg p(x')$. By the construction of g, g will map this prefix to 0, and therefore $\inf\{g(x') \mid x' \preceq x\} = 0 = g(x)$ and g induces a gray safety property.

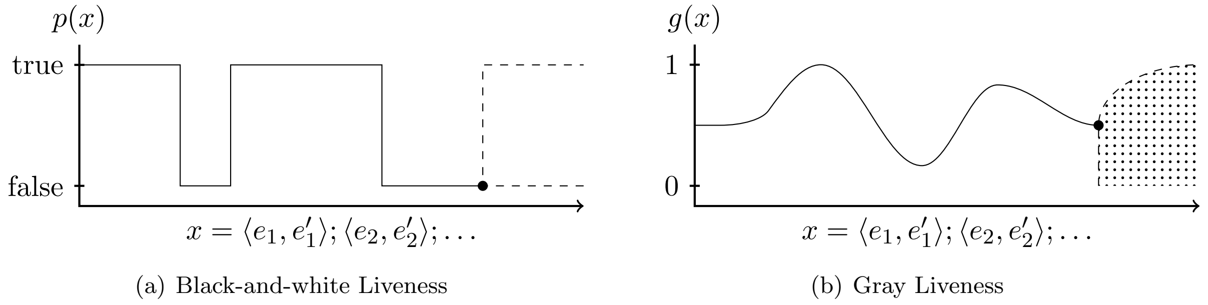


Figure 2.2. The security of traces as they proceed according to liveness properties. The security level is according to (a) a black-and-white liveness property and (b) a gray liveness property. The dotted lines and shaded area represent the possible security values of the executions' extensions. In all cases, less-than-fully-secure traces have more-secure extensions.

Conversely, suppose P is a non-safety property. Then there must exist some execution x such that either $p(x)$ and $\exists x' \preceq x : \neg p(x')$, or $\neg p(x)$ and $\forall x' \preceq x : p(x')$. In the former case, $\inf\{g(x') \mid x' \preceq x\} = 0 \neq g(x)$, and in the latter $\inf\{g(x') \mid x' \preceq x\} = 1 \neq g(x)$. Thus g does not induce a gray safety property. $\square$

2.1.2.2 Liveness

Black-and-white liveness properties require every finite trace to have a secure extension [1], as shown in Figure 2.2(a). A canonical example is the termination property, which requires traces to be finite (so every finite trace x has a secure extension, namely x).

Definition 8 ([1])**.** *A property p is* black-and-white liveness *iff* $\forall x \in \mathcal{E}^* : \exists y \succeq x : p(y)$.

Analogously, gray liveness properties require every finite trace to have a more-secure extension, with traces that are already 100% secure exempted (because a fully secure trace cannot have a more-secure extension). Figure 2.2(b) illustrates the requirement that, according to a gray liveness property, every imperfectly secure trace has a more-secure extension.

To formalize gray liveness, a new operator $\overline{>}$ is defined that behaves exactly like a greater-than operator ($>$), except that $1 \overline{>} 1$.

Definition 9. *A property g is* gray liveness *iff* $\forall x \in \mathcal{E}^* : \exists y \succeq x : g(y) \overline{>} g(x)$.

For example, a gray liveness property could map trace x to a security value based on the number n of resources acquired but unreleased in x; the security level might be $1-0.01n$ when $0 \leq n \leq 100$ and 0 when $n > 100$. This property gives traces a 1% security penalty for every unreleased resource. It is a gray liveness property because every finite, imperfectly secure trace has a more-secure extension (in which acquired resources are released). It is interesting to compare the usefulness of, and information provided by, this gray property with its black-and-white version, which simply says that traces are secure iff all acquired resources eventually get released.

As with safety, gray liveness is a proper generalization of black-and-white liveness, as demonstrated by the following proposition.

Proposition 4. *Let P be a black-and-white property with predicate p. Construct a gray function g as follows:*

$$g(x) = \begin{cases} 1, & \textit{if } p(x) \\ 0, & \textit{if } \neg p(x) \end{cases}$$

Then, P is a black-and-white liveness property iff g induces a gray liveness property.

Proof. First, it is proved that if P is a black-and-white liveness property, then g induces a gray liveness property. By the definition of a black-and-white liveness property, $\forall x \in \mathcal{E}^* : \exists y \succeq x : p(y)$. By the construction of g, $g(y) = 1$. Therefore, $\forall x \in \mathcal{E}^*, \exists y \succeq x : g(y) \gtrsim g(x)$, because $\forall r \in \mathbb{R}_{[0,1]}, 1 \gtrsim r$. Thus, g induces a gray liveness property.

Conversely, suppose that P is not a black-and-white liveness property. By Definition 8, there exists $x \in \mathcal{E}^* : \forall y \succeq x : \neg p(y)$. Because $x \succeq x$, $\neg p(x)$. By the construction of g, x and all of its extensions are mapped to 0. Thus g does not induce a gray liveness property. □

2.1.3 Hypersafety and Hyperliveness

Just as black-and-white properties can be categorized as safety or liveness, the same can be done for black-and-white policies. Using the term "hyperproperty" to mean "policy", then, hyperproperties can be categorized as hypersafety or hyperliveness [30]. Intuitively, the definitions of hypersafety and hyperliveness raise the definitions of safety and liveness from the level of executions (properties) to the level of sets of executions (policies).

The definitions of safety and liveness rely on the the $\preceq$ and $\succeq$ operators to indicate *executions* being prefixed or extended; definitions of hypersafety and hyperliveness will need to raise these operators to the level of *sets of executions*. This raising is accomplished by defining a *terminating target system* X—that is, a set of finite executions—to *prefix* another target system Y, written $X \sqsubseteq Y$, iff every execution in X is a prefix of some execution in Y. Formally, given $X \subseteq \mathcal{E}^*$ and $Y \subseteq \mathcal{E}^\infty$, $X \sqsubseteq Y$ iff $\forall x \in X : \exists y \in Y : x \preceq y$ [30, Footnote 13]. The $Y \sqsupseteq X$ relation is defined symmetrically.

2.1.3.1 Hypersafety

Black-and-white hypersafety raises black-and-white safety from the level of traces (executions) to the level of target systems (sets of executions) by requiring that target systems are secure iff all their prefixes are secure.

Definition 10 ([30])**.** *A policy P is* black-and-white hypersafety *iff* $\forall X \subseteq \mathcal{E}^\infty : P(X) = (\forall X' \sqsubseteq X : P(X'))$.

Notice the strong similarity between Definitions 10 and 6. Unsurprisingly, the following definition, which generalizes from gray safety to gray hypersafety, as well as from black-and-white hypersafety to gray hypersafety, has strong similarility to Definition 7.

Definition 11. *A policy G is* gray hypersafety *iff* $\forall X \subseteq \mathcal{E}^\infty : G(X) = \inf\{G(X') \mid X' \sqsubseteq X\}$.

The reasoning that gray hypersafety is a proper generalization of black-and-white hypersafety follows the reasoning used in Proposition 3 to show that gray safety is a proper generalization of black-and-white safety.

Following [30], it is also possible to define (black-and-white and gray) k-hypersafety by restricting the set X' to have at most k elements.

Definition 12 ([30])**.** *A policy P is* black-and-white k-hypersafety *iff* $\forall X \subseteq \mathcal{E}^\infty : P(X) = (\forall X' \sqsubseteq X : P(X') \vee |X'| > k)\}$.

Definition 13. *A policy G is* gray k-hypersafety *iff* $\forall X \subseteq \mathcal{E}^\infty : G(X) = \inf\{G(X') \mid X' \sqsubseteq X,\, |X'| \leq k\}$.

2.1.3.2 Hyperliveness

Black-and-white hyperliveness requires that all terminating target systems have secure extensions.

Definition 14 ([30])**.** *A* policy P *is* black-and-white hyperliveness *iff* $\forall X \subseteq \mathcal{E}^* : \exists Y \sqsupseteq X : P(Y)$..

Similarly, gray hyperliveness requires that all terminating target systems have more-secure extensions.

Definition 15. *A* policy G *is* gray hyperliveness *iff* $\forall X \subseteq \mathcal{E}^* : \exists Y \sqsupseteq X : G(Y) \gtrdot G(X)$.

Gray hyperliveness is a proper generalization of black-and-white hyperliveness by the same reasoning used in Proposition 4 to show that gray liveness properly generalizes black-and-white liveness.

2.1.4 Singleton Intersection of Safety and Liveness

In the black-and-white models, exactly one property is both safety and liveness: the "allow-all" property that considers every trace secure [1]. Similarly, exactly one policy is both hypersafety and hyperliveness: the policy that considers every target system secure [30]. The following theorems show that, analogously, exactly one property (policy) is both gray (hyper)safety and gray (hyper)liveness: the property (policy) that considers every trace (target system) perfectly secure.

Theorem 1. *The gray property $g(x) = 1$ is the only gray property that is both gray safety and gray liveness.*

Proof. First note that $g(x) = 1$ is trivially both a gray safety and a gray liveness property.

Now let g' be an arbitrary gray property that is both gray safety and gray liveness. For the sake of obtaining a contradiction, suppose there exists an execution x such that $g'(x) < 1$. If x is infinite, it must have a finite prefix whose security is also less than 1 because g' is gray safety; let x

instead refer to that prefix. Because g' is gray liveness, there exists $y \succeq x$ such that $g'(y) \gtrdot g'(x)$, so because $g'(x) \neq 1$, it must be that $g'(y) > g'(x)$. Also, because g' is gray safety, $g'(y)$ must equal $\inf\{g'(y') \mid y' \preceq y\}$. However, x is a prefix of y, so by the definition of infimum, $g'(y) \leq g'(x)$, which contradicts the earlier result that $g'(y) > g'(x)$. Thus, for all x, $g'(x) = 1$, meaning that g' must be g. □

Theorem 2. *The gray policy $G(X) = 1$ is the only gray policy that is both gray hypersafety and gray hyperliveness.*

Proof. First note that $G(X) = 1$ is trivially both a gray hypersafety and a gray hyperliveness property.

Now let G' be an arbitrary gray property that is both gray hypersafety and gray hyperliveness. For the sake of obtaining a contradiction, suppose there exists an target system X such that $G'(X) < 1$. Because G' is gray hypersafety, it must have a terminating target system prefix X' such that $G'(X') < 1$. Because G' is gray hyperliveness, there exists $Y \sqsupseteq X'$ such that $G'(Y) \gtrdot G'(X')$, so because $G'(X') \neq 1$, it must be that $G'(Y) > G'(X')$. Also, because G' is gray hypersafety, $G'(Y)$ must equal $\inf\{G'(Y') \mid Y' \sqsubseteq Y\}$. However, X' is a prefix of Y', so by the definition of infimum, $G'(y) \leq G'(X')$, which contradicts the earlier result that $G'(Y) > G'(X')$. Thus, for all X, $G'(X) = 1$, meaning that G' must be G. □

Figure 2.3 depicts the relationships between gray properties, black-and-white properties, and their subsets of safety and liveness properties. Notably, the gray sets subsume the black-and-white sets, and the intersection of safety and liveness is the black-and-white allow-all property.

2.1.5 Decomposition into Safety and Liveness

In the black-and-white models, every property p can be decomposed into properties p_s and p_l such that:

1. p_s is a black-and-white safety property,
2. p_l is a black-and-white liveness property, and

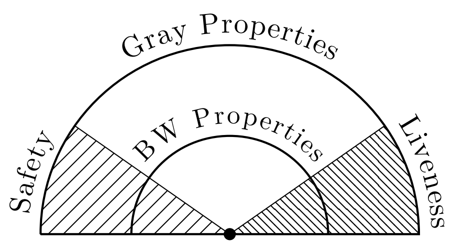


Figure 2.3. Relationships between gray and black-and-white properties. Also depicted are their subsets of safety and liveness properties. The central dot represents the intersection of safety and liveness, which only contains the property $g(x) = 1$.

3. $p(x) = (p_s(x) \wedge p_l(x))$.

In other words, every black-and-white property is the conjunction of a single safety and a single liveness property. This result appeared in [1], with alternative proofs appearing in [100, 64]. A similar result has been shown for decomposing policies into hypersafety and hyperliveness [30].

Theorem 3 shows that the gray model preserves this decomposition result.

Theorem 3. *Every gray property g can be decomposed into g_s and g_ℓ such that:*

1. *g_s is a gray safety property,*
2. *g_ℓ is a gray liveness property, and*
3. *$g(x) = min(g_s(x), g_\ell(x))$.*

Proof. Construct g_s and g_ℓ as follows, where *sup* refers to the supremum function.

$$g_s(x) = \begin{cases} \inf\{g_s(x') \mid x' \preceq x\} & \text{if } x \in \mathcal{E}^\omega \\ g(x) & \text{if } x \in \mathcal{E}^* \text{ and } \forall x' \succeq x : g(x') \leq g(x) \\ \sup\{g(x') \mid x' \succeq x\} & \text{otherwise} \end{cases}$$

$$g_\ell(x) = \begin{cases} 1 & \text{if } x \in \mathcal{E}^* \text{ and } \forall x' \succeq x : g(x') \leq g(x) \\ g(x) & \text{otherwise} \end{cases}$$

To establish that g_s is gray safety, it must be shown that for all $x \in \mathcal{E}^\infty$, $g_s(x) = \inf\{g_s(x') \mid x' \preceq x\}$. By construction, this constraint holds for all $x \in \mathcal{E}^\omega$. For finite executions, g_s ensures that security never increases as traces proceed by giving every finite trace x a security value that's greater than or equal to all of x's extensions. Hence, the safety constraint holds for all finite executions as well.

To establish that g_ℓ is gray liveness, it must be shown that for all $x \in \mathcal{E}^*$, $\exists y \succeq x : g_\ell(y) \gtrless g_\ell(x)$. Let x be a finite execution. If all extensions x' of x have security less than or equal to x (according to g), then $g_\ell(x) = 1$ and the liveness constraint is satisfied by letting $y = x$. On the other hand, if some extension x' of x has security greater than x (according to g), then $g_\ell(x) = g(x)$ and the liveness constraint is satisfied by letting $y = x'$ (where $g_\ell(x')$ must be at least $g(x')$, which is greater than $g(x) = g_\ell(x)$).

It remains to establish that $g(x) = min(g_s(x), g_\ell(x))$. If x is a finite trace then this result immediately follows from the definitions of g_s and g_ℓ. If x is an infinite trace, first observe that g_s assigns every prefix x' of x a security level of at least $g(x)$. Hence, g_s assigns $x \in \mathcal{E}^\omega$ a security level of at least $g(x)$, while g_ℓ assigns $x \in \mathcal{E}^\omega$ a security level of $g(x)$, which completes the proof.

□

As in the black-and-white models, the decomposition result in the gray model extends to policies, hypersafety, and hyperliveness.

Theorem 4. *Every gray policy G can be decomposed into G_s and G_ℓ such that:*

1. *G_s is a gray hypersafety policy,*
2. *G_ℓ is a gray hyperliveness policy, and*
3. *$G(X) = min(G_s(X), G_\ell(X))$.*

Proof. Construct G_s and G_ℓ as follows, where *sup* refers to the supremum function.

$$G_s(X) = \begin{cases} G(X) & \text{if } X \sqsubseteq X \text{ and } \forall X' \succeq X : G(X') \leq G(X) \\ \sup\{G(X') \mid X' \sqsubseteq X\} & \text{if } X \sqsubseteq X \text{ and } \exists X' \succeq X : G(X') > G(X) \\ \inf\{G_s(X') \mid X' \sqsubseteq X\} & \text{otherwise} \end{cases}$$

$$G_\ell(X) = \begin{cases} 1 & \text{if } X \sqsubseteq X \text{ and } \forall X' \sqsubseteq X : G(X') \leq G(X) \\ G(X) & \text{otherwise} \end{cases}$$

To establish that G_s is gray hypersafety, it must be shown that for all $X \subseteq \mathcal{E}^\infty$, $G_s(X) = \inf\{G_s(X') \mid X' \sqsubseteq x\}$. By construction, this constraint holds for all X such that $X \not\sqsubseteq X$ (i.e., when X is not a terminating target system). For terminating target systems, G_s ensures that security never increases as target systems extend by giving every terminating target system X a security value that's greater than or equal to all of X's extensions. Hence, the hypersafety constraint holds for all terminating target systems as well.

To establish that G_ℓ is gray liveness, it must be shown that for all $X \subseteq \mathcal{E}^*$, $\exists Y \sqsupseteq X$: $G_\ell(Y) \mathrel{\overline{>}} G_l(X)$. Let X be a terminating target system. If all extensions X' of X have security less than or equal to X (according to G), then $G_\ell(X) = 1$ and the liveness constraint is satisfied by letting $Y = X$. On the other hand, if some extension X' of X has security greater than X (according to G), then $G_\ell(X) = G(X)$ and the liveness constraint is satisfied by letting $Y = X'$ (where $G_\ell(X')$ must be at least $G(X')$, which is greater than $G(X) = G_\ell(X)$).

It remains to establish that $G(X) = min(G_s(X), G_\ell(X))$. If X is a terminating target system then this result immediately follows from the definitions of G_s and G_ℓ. Otherwise, first observe that G_s assigns every prefix X' of X a security level of at least $G(X)$. Hence, G_s assigns X a security level of at least $G(x)$, while G_ℓ assigns X a security level of $G(X)$, which completes the proof. $\square$

Table 2.1. Summary of the generalization from black-and-white to gray policies. The black-and-white definitions are taken from [101, 1, 37, 30]. As a reminder, the $\bar{>}$ operator behaves like greater-than ($>$), except that $1 \bar{>} 1$.

policy	◩	$P : 2^{\mathcal{E}^\infty} \rightarrow \{false, true\}$
	▨	$G : 2^{\mathcal{E}^\infty} \rightarrow \mathbb{R}_{[0,1]}$
property	◩	$\exists p : \forall X \subseteq \mathcal{E}^\infty : P(X) = (\forall x \in X : p(x))$
	▨	$\exists g : \forall X \subseteq \mathcal{E}^\infty : G(X) = \inf\{g(x) \mid x \in X\}$
safety	◩	$\forall x \in \mathcal{E}^\infty : p(x) = (\forall x' \preceq x : p(x'))$
	▨	$\forall x \in \mathcal{E}^\infty : g(x) = \inf\{g(x') \mid x' \preceq x\}$
liveness	◩	$\forall x \in \mathcal{E}^* : \exists y \succeq x : p(y)$
	▨	$\forall x \in \mathcal{E}^* : \exists y \succeq x : g(y) \bar{>} g(x)$
hypersafety	◩	$\forall X \subseteq \mathcal{E}^\infty : P(X) = (\forall X' \sqsubseteq X : P(X'))$
	▨	$\forall X \subseteq \mathcal{E}^\infty : G(X) = \inf\{G(X') \mid X' \sqsubseteq X\}$
hyperliveness	◩	$\forall X \subseteq \mathcal{E}^* : \exists Y \sqsupseteq X : P(Y)$
	▨	$\forall X \subseteq \mathcal{E}^* : \exists Y \sqsupseteq X : G(Y) \bar{>} G(X)$

2.1.6 Summary

Table 2.1 summarizes the gray definitions and compares each with its black-and-white counterpart.

2.2 Silhouettes and Their Judges

The gray model separates specifications of how secure target systems are (gray policies) from specifications of how secure users require their systems to be (silhouette judges). In the safe analogy of Chapter 1, silhouette judges input a safe's security rating and output a buy or don't-buy decision. In other words, silhouette judges input a *silhouette*—a distillation of a safe's characteristics into a security value—and output a no/yes decision to indicate whether that silhouette is acceptably secure.

2.2.1 Silhouettes

Thus, silhouette judges, as their name implies, judge silhouettes, by outputting a no/yes (or false/true) to indicate whether a given silhouette is acceptably secure.

In the gray model, a *silhouette* represents the shape of a trace's (or target system's) security. For example, the plots shown in Figures 2.1 and 2.2 illustrate silhouettes of traces—the plots abstract from the events of the underlying traces to provide only the shape of the security values achieved as the traces proceed.

Silhouettes can be formalized in many ways. For generality, the key requirement is to encode a trace's (or target system's) evolution of security values.

For example, the silhouette of a trace x according to property g can be formalized as a function s that takes a natural number n, or a special ∞ symbol, as input and returns the security (according to g) of x's n-length prefix, or the security of x itself if s's input is ∞. With this formalization, silhouettes are partial functions; e.g., the silhouette of the empty trace is undefined for all inputs $n > 0$.

With this formalization, *the silhouette of trace x according to gray property g* is the partial function $s_{x,g} : (\mathbb{N} \cup \{\infty\}) \to \mathbb{R}_{[0,1]}$, such that:

$$s_{x,g}(n) = \begin{cases} g(x) & \text{if } n = \infty \\ g(x') & \text{if } x' \text{ is the } n\text{-length prefix of } x \end{cases}$$

Because target systems may be infinite sets of infinite-length traces, silhouettes of target systems are more complicated than those of individual traces. Rather than mapping natural numbers to security values, target-system silhouettes could map real numbers to security values. In this case, the real number can encode which parts of the target system's traces to evaluate the security of.

For example, a silhouette for target system X could interpret an input like 0.192939..969109119... as identifying the set of traces containing the 1-length prefix of X's first execution (ordered lexicographically), the 2-length prefix of X's second execution, and so on, with each prefix length delimited by a 9 and written in base-7. Under this encoding, the target-system silhouette could interpret a 7 (8) appearing before the i^{th} 9 in an input real number as indicating exclusion of the (inclusion of the whole) i^{th} execution in X.

With such a formalization, the *silhouette of target system X according to gray policy G* is the partial function $S_{X,G} : \mathbb{R} \rightarrow \mathbb{R}_{[0,1]}$, such that:

$$S_{X,G}(r) = G(X'), \text{ where } r \text{ encodes } X' \text{ with respect to } X$$

2.2.2 Silhouette Judges

Silhouette judges are simply predicates over silhouettes. A judge therefore acts as the final, black-and-white decision maker, determining whether a trace or target system is acceptably secure. Importantly, judges base their decisions on *silhouettes* of traces or target systems, not on the traces or target systems themselves, as is done in black-and-white models.

For example, a silhouette judge could forbid all trace silhouettes whose security ever drops below a certain minimum threshold. This sort of silhouette judge specifies a user's minimum security requirement, such as "traces must always be at least 80% secure".

Another silhouette judge might forbid all silhouettes whose "final" security value (obtained by inputting ∞ or 0.8989898... to the given silhouette) is greater than 0. Such a judge might be used by high-performance systems researchers to require the complete insecurity of their executions.

More interesting judges can also be defined. For example, it may be reasonable to allow executions to occasionally behave less securely, provided they are usually more secure. Such a judge might be satisfied by exactly those silhouettes having a rolling average of security above a given threshold. Other judges could be satisfied by exactly those silhouettes that never dip below a desired threshold for more than k consecutive exchanges.

Theorem 5 states that combining a gray property g with a trace-silhouette judge j produces a unique black-and-white property p, but, on the other hand, every black-and-white property can be decomposed into uncountably many different gray-property, trace-silhouette-judge pairs. Theorem 6 states a similar result for black-and-white policies and gray-policy, trace-system-silhouette-judge pairs.

Theorem 5. *There is a one-to-uncountably-many correspondence between black-and-white properties p and pairs of gray properties and trace-silhouette judges (g, j) such that $\forall x \in \mathcal{E}^\infty : p(x) \Leftrightarrow j(s_{x,g})$.*

Proof. Every gray-property, trace-silhouette-judge pair (g, j) is equivalent to exactly one black-and-white property p; otherwise, there must be some execution x whose silhouette according to g, $s_{x,g}$, both satisfies and dissatisfies j, a contradiction.

It remains to show that every black-and-white property can be expressed by an uncountable number of gray-property, silhouette-judge pairs. Given black-and-white property p and arbitrary $r \in \mathbb{R}_{[0,1]}$, construct a gray property g_r and silhouette judge j_r as follows:

$$g_r(x) = \begin{cases} r & \text{if } p(x) \\ 0 & \text{otherwise} \end{cases}$$

$$j_r(s) \Leftrightarrow (s(\infty) = r)$$

By construction, $p(x) \Leftrightarrow j_r(s_{x,g_r})$. Because there are uncountably many values of r, there are uncountably many pairs of (g_r, j_r) equivalent to p.

□

Theorem 6. *There is a one-to-uncountably-many correspondence between black-and-white policies P and pairs of gray policies and target-system-silhouette judges (G, J) such that $\forall X \subseteq \mathcal{E}^\infty : P(X) \Leftrightarrow J(S_{X,G})$.*

Proof. Every gray-policy, target-system-silhouette-judge pair (G, J) is equivalent to exactly one black-and-white policy P; otherwise, there must be some target system X whose silhouette according to G, $S_{X,G}$, both satisfies and dissatisfies J, a contradiction.

It remains to show that every black-and-white policy can be expressed by an uncountable number of gray-policy, silhouette-judge pairs. Given black-and-white policy P and arbitrary $r \in \mathbb{R}_{[0,1]}$, construct a gray policy G_r and silhouette judge J_r as follows:

$$G_r(X) = \begin{cases} r & \text{if } P(X) \\ 0 & \text{otherwise} \end{cases}$$

$$J_r(S) \Leftrightarrow (S(0.898989...) = r)$$

By construction, $P(x) \Leftrightarrow J_r(S_{X,G_r})$. Because there are uncountably many values of r, there are uncountably many pairs of (G_r, J_r) equivalent to P. $\square$

These theorems demonstrate the increased expressiveness of gray policies, properties, and silhouette judges, compared to black-and-white policies and properties.

CHAPTER 3

DEFINING GRAY POLICIES

This chapter[1] presents several examples of gray policies to demonstrate how gray policies can be constructed in practice. At a high-level, two methods of constructing gray policies are explored:

1. leveraging existing works on security metrics, which seek to quantify the security of programs and/or executions (Section 3.1).
2. converting black-and-white security policies into usefully quantified gray policies (Section 3.2).

While gray policies can certainly be built in many other ways, these two approaches leverage existing research to construct interesting and useful policies.

3.1 Gray Policies from Security Metrics

Security metrics, which seek to quantify various aspects of security, are a natural source to rely on when constructing gray policies. In addition, security metrics in general benefit from the gray model, as the gray model serves as a unifying framework for disparate approaches to security metrics. The disparate approaches include:

1. using greater values to indicate higher levels of security (e.g., [36]),
2. using greater values to indicate lower levels of security (e.g., [73]),
3. limiting security values to a bounded range (e.g., [25]),
4. limiting security values to a range bounded only on the lower side (e.g., [63]), and

[1]Portions of this chapter were previously published in [96]. Permission to use this material is in Appendix A.

Table 3.1. Examples of functions that can normalize security metrics to the range of $\mathbb{R}_{[0,1]}$. Variable x denotes the output of the security metric, constants A and B denote the metric's minimum and maximum values (when applicable), and constant C denotes a positive number (C affects how quickly the functions converge to absolute security or insecurity).

	bounded	lower bounded	unbounded
higher values represent higher security	$y = \frac{x-A}{B-A}$	$y = \frac{x-A}{x-A+C}$	$y = 0.5 + \frac{\tan^{-1}(C * x)}{\pi}$
higher values represent lower security	$y = \frac{B-x}{B-A}$	$y = \frac{C}{x-A+C}$	$y = 0.5 + \frac{\tan^{-1}(-C * x)}{\pi}$

5. placing no bounds on the range of security values (e.g., [9]).

In contrast to black and white models, all of these approaches can be encoded in the gray model.

Encoding these disparate approaches to security metrics in the gray model provides the benefit of consistency. In the gray model, security consistently ranges between 0 and 1, and for a fixed policy or property, greater security values consistently indicate higher security.

Table 3.1 shows several example functions that can be used to encode security metrics as gray policies. The arctangent functions shown in this table can be used to normalize metrics having an unbounded range because the arctangent's domain is all real numbers, and its output monotonically increases over the range $(-\frac{\pi}{2}, \frac{\pi}{2})$. The arccotangent function ($\cot^{-1}$), and many others, could be used instead.

Using the functions presented in Table 3.1, security metrics from domains as varied as access control, noninterference, privacy, integrity, and network security can be encoded as a gray policy or property; the remainder of this section gives examples from each of these categories.

3.1.1 Access Control

Access control policies focus on ensuring that resources are only granted to authorized users. Typical black-and-white access control policies include access control lists [98], role-based access control [42], multi-level systems (e.g., the Bell-Lapudala [13] model), and the Chinese Wall [19].

Access control gray policies can be created by leveraging existing work on quantifying access control. For example, multi-level systems have been considered that view access rights as a quantitative attribute [26, 25]. To demonstrate, a user could be granted 80% of the right to access secret information; this represents a 20% risk that the user will disclose the secret information to an unauthorized entity. A gray function could be constructed for this kind of model by calculating the cumulative risk of an unauthorized disclosure as an execution proceeds and access to secret information is granted.

$$g(x) = \prod_{\langle e,e' \rangle \in x} Risk(\langle e, e' \rangle)$$

This gray policy is gray safety; as risk values are probabilities between 0 and 1, the overall risk of an execution can only decrease as it proceeds.

A similar construction of a gray policy can be used to express more recent work which models a distributed system with a central policy oracle being enforced locally by several policy enforcers [80]. Over time, the policy enforcers learn to predict how the oracle will respond to requests. If a policy enforcer is confident enough in its prediction, it can choose to not consult with the oracle. However, the predictions are uncertain, so the local enforcer might forbid a valid request, or grant an invalid one. The risk introduced by each local decision can be calculated by taking into account the utility of the request, and the potential damage of a false characterizations [80].

Other approaches to quantifying access control include measuring, for a specific resource, the "strength" of a user's (black-and-white) authorization (quantified by counting how many ways the user could prove authorization), or how close a user is to being authorized (quantified by counting how many credentials the user lacks) [62]. These metrics could be used to construct a gray function that maps executions to their "worst" resource grant, where grants for stronger authorizations are considered more secure while grants for more distant inauthorizations are considered less secure; if the execution has no worst resource grant, the greatest lower bound could be used.

To construct this gray policy, denote as $\Delta Cred$ the distance by which a user's credentials either surpass or fail to reach the required authorization; $\Delta Cred$ is 0 when the user exactly meets the requirements, positive when the user has more than enough credentials, and otherwise is negative.

$$g(x) = \inf \left\{ 0.5 + \frac{\tan^{-1}(\Delta Cred(\langle e, e' \rangle))}{\pi} \mid \langle e, e' \rangle \in x \right\}$$

This construction examines each exchange in an execution in isolation, and determines the relative strength (or weakness) of the credentials involved in the exchange, relative to the required authorization. As expected, an execution's security will be mapped to its weakest authorization, or to the greatest upper bound if the execution does not have a weakest authorization. By construction, this gray property is gray safety; executions which include weak resource grants cannot be made more secure.

Still other access control systems involve users earning "reputation", a quantitative score that can be used to determine which authorizations a user should be granted [9, 8, 114, 111, 112]. For example, a website might require a user to have at least 10 video reputation, or 50 comment reputation, before allowing the user to flag a video as inappropriate. More generally, reputation policies are boolean combinations of (Category, Threshold) pairs (e.g., `OR((Video, 10), (Comment, 50))`). The degree to which a user U satisfies a specific threshold T in category Cat could be quantified as follows:

$$d(U, (Cat, T)) = 0.5 + \frac{\tan^{-1}(U.Cat - T)}{\pi}$$

With this construction, a user that is just barely meets the threshold has a degree of 0.5; users with higher reputations get mapped asymptotically closer to 1, while users with lower reputations get mapped asymptotically closer to 0. Then, the degree to which a request satisfies a reputation policy could be computed by replacing all `OR` functions with max and all `AND` functions with min. Gray properties could then grade executions based on the degree to which individual reputation policies are satisfied.

3.1.2 Noninterference

Noninterference policies place restrictions on how trusted (secret) inputs to an application (e.g., insider trading information) can influence its untrusted (public) outputs (e.g., buy/sell operations). When noninterference policies are satisfied, it should be impossible for an attacker to learn anything about an application's trusted inputs by only observing its untrusted outputs [16, 47, 76].

In some applications, noninterference can be too strict of a requirement; the classic example is of a password checker, which leaks knowledge about the (trusted-input) password database when a wrong password is given (namely, that the actual password is *not* the given password) [29]. Thus, a (black-and-white) noninterference policy must forbid the password-checker from running.

As such, work has been done to relax noninterference by allowing an application to leak *some* knowledge of the trusted inputs [24, 104, 17, 29, 6, 4, 81, 41, 3, 82, 74]. Classically, leaked knowledge is measured in *bits*, where each bit effectively doubles an attacker's chance of correctly guessing the trusted inputs, although more recent work has measured leakage as an arbitrary "gain" of the attacker [4, 74]. Several methods for computing the leakage of a program have been presented, such as:

1. Calculating the ratio of an attacker's a priori and a posteriori uncertainty [104, 17, 6, 4, 3, 82]. In this context, uncertainty can be quantified using information-theoretic *entropy* (e.g., Shannon entropy, Rényi entropy). Before the execution begins, an attacker has some assumption of what the trusted input could be. For example, if the trusted input is an encryption key, the attacker could assume that all 128-bit values are equally probable; if the trusted input is a password, the attacker might consider some passwords as more likely than others. As an execution progresses, the initial assumption becomes more and more refined as parts of the trusted input are eliminated or confirmed. An equivalent metric that takes the *difference* of the a priori and a posteriori uncertainty has also been investigated [17].

2. Calculating the difference between an attacker's a priori and a posteri accuracy [29]. Here, accuracy is the distance between the actual trusted input distribution and what the attacker

believes to be the trusted input distribution, quantified as the relative entropy between the two.

Regardless of which information leakage method is used, the number of bits leaked by an execution (or the gain of an adversary) could be used as the basis for a gray function. Let L be a method for computing the leakage of an execution and define gray function g as follows:

$$g(x) = 0.5 + \frac{\tan^{-1}(-L(x))}{\pi}$$

When L is based on an uncertainty metric and the attacker has perfect knowledge of the program's behavior, the attacker's uncertainty can only decrease (i.e., the information leakage can only increase) as a program executes [82]. In this special case, g induces a gray safety property; no extension of an execution can make the attacker more uncertain.

However, when L is based on accuracy, or if the attacker does not have perfect knowledge of the program's behavior, it is possible for an execution to have negative information flow [82, 29]. In such cases, an extension can be more secure than the execution it extends, and therefore g does not induce a gray safety property.

Other notions of black-and-white noninterference compare different executions of a target system that differ in their trusted inputs and require that untrusted outputs be unaffected by these changes in trusted inputs. Such notions place noninterference firmly outside of the realm of properties, because the overall security of a target system depends on relationships between different executions, rather than solely the executions themselves. For example, a target system might be permitted to contain a certain trace if it does not contain another particular trace.

Unfortunately, there does not seem to be a security metric that quantifies the degree to which target systems satisfy such notions of noninterference. If such a metric existed, it would serve as an example of how to construct a gray non-property policy.

3.1.3 Integrity

Dual to information leakage is information integrity, which forbids untrusted inputs from affecting trusted outputs. When properly enforced, an attacker should be incapable of influencing the trusted outputs of an application. A classic example of an integrity policy is Biba's integrity model [14], in which trusted subjects cannot read untrusted data, and untrusted subjects cannot write to trusted data.

Just as noninterference prevents a password-checking program, strict models of integrity may also forbid otherwise reasonable programs. For example, if the password database of an operating system is considered trusted, the Biba integrity policy would forbid an untrusted user from changing his or her password. Thus, it may be desirable to relax integrity policies and allow some amount of integrity loss to occur.

Relatively little work has been done on quantifying integrity loss. Clarkson and Schneider [31] quantify the damage to integrity as "corruption", measured in bits. Corruption can come from either the contamination of trusted outputs by untrusted inputs, or the suppression of trusted inputs before they are output. For example, if the trusted output is the result of a bitwise `OR` of a trusted input and a random, untrusted input then contamination has occurred, whereas outputting a truncated version of an input is suppression.

Let $Corruption(x)$ denote the corruption exhibited by an execution x (measured in bits), and let $|TOut(x)|$ denote the number of bits in x's trusted outputs. A gray function that maps executions to their percent integrity could be defined as follows.

$$g(x) = \begin{cases} \inf\{g(y) | y \preceq x\} & \text{if } |TOut(x)| \text{ is infinite} \\ \frac{Corruption(x)}{|TOut(x)|} & \text{otherwise} \end{cases}$$

This gray function induces a gray liveness property because any finite execution can be made more secure by outputting trusted inputs.

Like noninterference, integrity is often viewed as a non-property policy, but no quantified security metrics have been formalized to measure this kind of idea.

3.1.4 Privacy and Anonymity

The advent of big data and the growing use of technology in historically repressive areas has renewed interest in privacy, where entities choose which information about themselves are available to others. Security policies that enforce privacy limit or prevent the unauthorized or unintentional disclosure of user information.

Some privacy policies are enforced by the users of a service. For example, suppose a user wishes to use a search engine, but does not want the search engine to learn about his or her interests. The search engine, on the other hand, wishes to build a profile of the user's interests for its own use, or to be sold at a later time. Several tools have been proposed e.g., [58, 27]) that add "chaff" to users' interactions with the search engine; the search engine, unable to distinguish between the actual query and the chaff, has a more difficult time constructing user profiles. The methods used to quantify the privacy provided by such tools (e.g., [46, 92]) could be used to construct privacy gray functions.

Other privacy policies are enforced by the service being used. For example, an organization might monitor its employees and calculate the probability that an employee is violating the (black-and-white) privacy policy of the organization [5]. To calculate this probability, a dynamic Bayesian network could be constructed that models the normal, policy-obeying behavior of the employee. The Bayesian network encodes relationships such as "if the employee spends more than 5 minutes on a case, it is more likely that the privacy policy is being violated" or "if the behavior of the employee was normal yesterday, it is more likely that the employee will behave normal today". A gray function could be constructed that simulates a dynamic Bayesian network for each employee in the given execution and returns the total chance that all employees are following the privacy policy.

$$g(x) = \prod_{emp \in Employees} (\text{probability that } emp \text{ 's actions in } x \text{ follow the privacy policy})$$

An interesting subset of privacy policies enforce anonymity to attempt to prevent outside observers from knowing which users are performing various actions. Classic examples of mechanisms that enforce anonymity are the Dining Cryptographers [23] and Crowds [97] protocols. Several methods have been proposed for evaluating and quantifying the anonymity provided by protocols such as these; the remainder of this subsection provides examples of how such metrics could be used to create gray anonymity policies.

Simple metrics for measuring anonymity include counting the number of users of the protocol [23], or calculating the maximum chance that an attacker can link a user with a message created by the user [54]. However, these metrics do not provide many guarantees [36]. For example, the second of these does not differentiate between when one user is most likely the creator amongst a large number of users, and when one user is equally likely amongst a small number of users; it can be argued that the latter is a more secure situation.

More advanced metrics can be constructed using information-theoretic entropy [36, 103, 33, 22, 32, 49]. At a high level, these approaches assume that, after observing a trace of messages, the attacker has some probability distribution of which message belongs to which person. In the worst case, the attacker knows with absolute certainly who sent each message, and the entropy of the probability distribution is 0. In the best case, each message could be attributed to each user with equal probability, and the entropy of the probability distribution is maximized.

Let $H(APD|x)$ be the entropy of the attacker's probability distribution after observing an execution x, and $H(UPD|x)$ be the best-case entropy for execution x. Then, a gray function can be constructed as follows.

$$g(x) = \frac{H(APD|x)}{H(UPD|x)}$$

Another measure of anonymity provided by a protocol comes from counting how many perfect matchings there are in a bipartite graph where one partition represents input messages and the other output messages [40]. This approach assumes that an observer can use timing information to eliminate possible input/output message combinations. For example, if a protocol ensures that a message will get sent within some time limit t, then an output message could only have come from messages input in the last t units of time.

In a bipartite graph, the number of perfect matchings can be calculated by taking the permanent of its adjacency matrix. In the best case where every output message could be any input message, the number of perfect matchings is $n!$, where n is the number of messages. In the worst case, there is only 1 perfect matching.

Assume that $M(x)$ returns the set of (input and output) messages in an execution x; if there are more input messages than output messages, $M(x)$ adds "dummy" output messages that occur at the same time as the last message in x. Then, if *per* calculates the permanent of the adjacency matrix imposed by a set of messages, Formula 2 of [40] can be used to help define a gray function:

$$g(x) = \begin{cases} 1 & \text{if } |M(x)| = 0 \\ 0 & \text{if } |M(x)| = 2 \\ \inf(\{g_{Anon}(y) | y \preceq x\}) & \text{if } M(x) \text{ is infinite} \\ \frac{\log(per(M(x)))}{\log(\frac{|M(x)|}{2}!)} & \text{otherwise} \end{cases}$$

Note that the final case of this construction will always result in a number between 0 and 1; it is computing the ratio between the number of perfect matchings in the message matrix and the maximum possible number of perfect matchings for a matrix of that size. Thus, the closer a matrix is to having as many perfect matchings as possible, the higher its security will be.

3.1.5 Network Security

Network security policies are meant to ensure the overall security of a network system. Typical black-and-white policies include firewall policies, routing policies, etc. that try to ensure that the network is only used for valid traffic. A large variety of network security metrics have been proposed; this subsection explores how some of these metrics can be used to construct gray policies.

One high-level approach could be to repeatedly simulate a network against a random source of attacks [21, 116]. The ratio of successful attacks to the number of simulations is a natural gray policy, and can be directly computed using existing techniques.

In another work, a network is represented as a state machine, where each state is a network configuration and edges represent different exploits available to attacks, weighted by how challenging it is to use or create the exploit [91]. Some configurations are marked as “critical”, and an attack succeeds when it places the network in a critical state.

To judge the overall security of the network, the state machine is analyzed to find the smallest set of exploits (by weight) that can move the network from its current state to a critical state. This set represents the so-called “weakest adversary” that can penetrate the network. Consider a function WA that, when given an execution, computes the cumulative weight of the weakest adversary for the state machine, assuming that the initial state is the one reached by simulating the network's state machine on the given execution as input. A gray function can then be constructed from WA:

$$g(x) = \frac{WA(x)}{1 + WA(x)}$$

This gray property is neither safety nor liveness; it is possible that a fully-compromised network cannot be made safe again, but partially compromised networks might be made more secure by undoing the exploits performed by the adversary.

Alternatively, a gray security function could be constructed to calculate the chance that an attack is occurring. One approach is to construct an attack graph from the network graph. In the attack graph, network vulnerabilities are vertexes and edges represent preconditions [44]. Each vertex is annotated with a chance of success using existing vulnerability databases, and some vertices are marked as critical. The chance that an attack occurs can then be calculated directly by finding the most likely path from the attack graph's initial states (i.e., the vertices that have no preconditions) to a critical state. In this model, a gray function can be constructed in a manner similar to the previous construction by recomputing this chance of attack success under the assumption that the initial states are those reached by an already-executed prefix x from the original input states.

AssetRank [99], another attack-graph approach, instead analyzes attack graphs using a modified version of the PageRank algorithm [90] to identify which vulnerabilities contribute most to a system's insecurity. Using AssetRank, a gray policy can be constructed that maps systems with particularly insecure components to lower and lower values as follows.

$$\mathcal{G}(X) = \frac{\text{mean asset rank of } X\text{'s attack graph}}{\text{maximum asset rank of } X\text{'s attack graph}}$$

Other network security policies instead compute the chance that some goal will be satisfied. For example, one approach constructs a "goal graph" where vertices are either goals (e.g., database contents are confidential), events (e.g., an attacker finds a SQL-injection vulnerability), or treatments (e.g., replacing all database accesses with parameterized queries) [7]. In this graph, vertices are labeled with a chance of success and a chance of failure, and edges between vertices describe relationships such as "replacing all database accesses with parameterized queries make it less likely for an attacker to find a SQL injection vulnerability" and "an attacker finding a SQL-injection vulnerability makes it more likely that database contents will not be confidential".

Analysis of the resulting graph can derive the overall chance that a particular goal is satisfied. Other approaches use simpler tree structures to calculate this probability [10, 28]. Regardless of how the probability is computed, it can be used as the basis of a gray policy without modification.

Yet another network-security metric arises from intrusion-detection mechanisms, which observe network traffic to detect ongoing attacks. Typically, an intrusion-detection mechanism computes the probability that an attack is occurring, and raises an alert when this probability raises above some threshold. While there is some inherent uncertainty associated with these probabilities [88], they could nonetheless be used to construct useful gray policies.

For example, one intrusion-detection mechanism augments a network with a number of agents that monitor traffic [50]. Each agent has an internal Bayesian network that calculates the chance that a specific attack is occurring (e.g., one agent might be configured to detect denial-of-service attacks, while another might be configured to detect IP spoofing, etc.). As the network is used, these agents report to a centralized agent that has its own Bayesian network for computing the overall chance that an attack is occurring. A gray function could be constructed by simulating the given execution in this agent-based model and returning the output of the centralized agent.

3.1.5.1 Mean Time to Security Failure

In reliability theory, mean time to failure is a common measure of how reliable a system is that represents the average length of time between system failures. This concept was extended by quantified security researchers into mean time to security failure, which represents the average amount of time required of an attacker to cause a security failure [66]. Several methods to calculate this value for a program have been proposed, including:

1. Constructing an attack graph that shows that states involved in a successful attack (e.g. reconnaissance, penetration, escalation), and computing or estimating the amount of effort required to transition from each state to the next. This graph can then by analyzed as a Markov chain, and the mean time to security failure computer [63, 77].

2. Constructing a graph where nodes represent system states. The effort required to transition the system from one state to another is computed, and the states are classified as either secure, vulnerable, or compromised. This graph is then analyzed as a Markov chain, and the average effort taken to get the system into a compromised state is computed [69, 68, 51].

3. Constructing a graph where nodes represent a set of user privileges. An edge from one node to another represents the capability of the first user to assume the privileges of the second user, either legitimately, from guessing the second user's password, or from using some exploit. Each of these edges is weighted based on the difficulty of the transition, and then the graph is analyzed as a Markov chain, and the average effort required to go from normal privileges to root privileges is calculated [34, 87].

4. Constructing a formula that attempts to calculate mean-time-to-security failure directly [78]. This formula has several parameters, including the time an exploit takes to run, the chance an existing exploit can succeed, the time required to construct a new exploit, the chance that a newly-constructed exploit will fail, and the time that it takes to find new vulnerabilities to exploit. Some of these parameters, such as the time it takes to construct a new exploit, are dependent on the skill of the proposed attacker, whereas others, such as the

chance that a constructed exploit will fail, are dependent on the security of the system being attacked.

Regardless of how the mean time to security failure is computed, translating the resulting number to a gray policy is straightforward. Consider the following policy, where $MTTSF$ is the mean time to security failure measured in hours:

$$\mathcal{G}(X) = \frac{MTTSF(X)}{1 + MTTSF(X)}$$

Based on this policy, programs that take no effort to breach are completely insecure, while programs that have a higher mean time to security failure will be mapped to values closer and closer to 1. Note that there is an implicit assumption in this model that the mean time to software failure is finite; that is, a breach will happen with enough effort, and as such no program is completely secure (i.e. mapped to the value 1).

3.1.5.2 Attackable Surface Area

Another common network security metric is attack surface area [57, 56, 107, 73, 72, 45], which is based on the principle that an attacker can only breach the security of a system if he or she can interact with it. Furthermore, the capabilities of the attacker are limited by the capabilities of the part of the system that he or she is interacting with.

The various attack surface area metrics quantify the security of a system by measuring the number of ways an attacker can interact with the system, while weighting each of these interactions by the damage they are capable of causing. Regardless of exactly how the attack surface area is calculated, the existing metrics can be expressed as gray policies in the following manner:

$$\mathcal{G}(X) = \frac{1}{1 + \text{attack surface area of X}}$$

According to this policy, any program that does not accept any input is considered perfectly secure, and programs with higher attack surface area are mapped to lower and lower values.

Alternatively, a gray function could be constructed based on an attack surface area metric for individual executions. For example, a gray function could lower the security of an execution once it opens a socket, only to be (fully or partially) restored when the socket is closed.

3.2 Graying Black-and-white Policies

Gray policies can also be created from existing black-and-white policies. For example, a gray policy $G(X)$ could be created by quantifying how well the given target system X obeys a particular black-and-white policy. This technique has already been used in this dissertation's examples: black-and-white policies might require all array accesses to be checked or all acquired resources to be released; these policies were grayed by penalizing target systems based on how far their traces deviate from ideal traces. A similar idea is used with cost-aware enforcement [38], where a cost, or penalty, can be assigned to certain exchanges.

Another approach to graying considers the overall security achieved by permitting some "insecure" executions to be run and/or denying some "secure" executions from being run [39].

Gray policies could be defined based on a similar idea: Given a black-and-white property of interest p, $G(X)$ might be defined as the product of:

- the probability that a randomly selected element of X satisfies p—*such a probability measures the* soundness *of X with respect to p*—and
- the probability that a randomly selected element of $\{x \mid p(x)\}$ is in X—*such a probability measures the* completeness *of X with respect to p.*

Following [71], these probabilities could be weighted by the likelihood of traces to actually be observed (due to nonuniform input distributions and target-system functionality, some traces may be observed much more frequently than others). Therefore, when calculating $G(X)$ in terms of the soundness and completeness probabilities defined above, one might choose traces not from uniform distributions, but instead with the more-likely-to-be-observed traces more likely to be chosen.

CHAPTER 4

PREVENTING INJECTION ATTACKS

According to multiple sources, the most commonly reported software attacks are injection attacks, including SQL injections, cross-site scripting, and OS-command injections [79, 85, 89]. To demonstrate the value of the gray model, this chapter[1] explores how gray policies and silhouette judges can be formed to mitigate code-injection attacks.

4.1 Examples of Injection Attacks

Applications vulnerable to injection attacks generate output programs based on untrusted inputs. By providing a malicious input, an attacker can cause the application to output a malicious program. A classic example involves a simple web application for a bank; the application inputs a password and returns the balance of accounts with the given password. On a typical, benign input such as `123456`, the banking application outputs the following program (throughout this chapter, input symbols injected into the output program are underlined).

$$\texttt{SELECT balance FROM accts WHERE pw=`}\underline{\texttt{123456}}\texttt{'}$$

Unfortunately, if this application does not validate its input, it can be manipulated into creating malicious programs, such as the following.

$$\texttt{SELECT balance FROM accts WHERE pw=`}\underline{\texttt{' OR 1=1 --}}\texttt{'}$$

This output program circumvents the password check and returns the balances of all accounts because (1) the 1=1 subexpression is a tautology, making the entire `WHERE` clause true, and (2) in SQL, the `--` sequence begins a comment, which removes the final apostrophe and makes

[1]Many of the ideas and much of the text in this chapter were previously published in [94, 95]. Permission to use this material is in Appendix A.

the program syntactically valid. Because some of the injected symbols are code symbols (i.e., disjunction and equality operators), this is an example of a *code*-injection attack.

Due to their prevalence, much research has been performed to define code-injection attacks formally (e.g., [94, 108, 15, 52, 83, 113]). Many additional papers have described tools for detecting and preventing code-injection attacks (e.g., [59, 20, 60, 67, 18, 55, 86, 93, 105]).

Interestingly, a related class of attacks exists but has not yet, as far as we're aware, been explored. These related attacks have many of the symptoms of code-injection attacks but don't involve injecting *code*. Because these attacks are performed by injecting noncode symbols, we call them *noncode-injection attacks*. Although noncode-injection attacks don't involve injecting malicious code, they may cause other parts of the output program to execute maliciously. For example, the following web app[2] is vulnerable to noncode-injection attacks.

```
$attackerControlledString = input();
$code = ‘‘\\$data = ‘$attackerControlledString’; ’’
      . ‘‘securityCheck(); \\$data .= ‘&f=exit#’;\n f();’’;
eval($code);
```

On a benign input such as `Hello!`, this application outputs the following.

```
$data = ‘Hello!’; securityCheck(); $data .= ‘&f=exit#’;\n f();
```

This output program sets `$data` to a value, calls the `securityCheck()` function, appends a string to `$data`, and then invokes the `f` function. However, if an attacker enters the string `\`, the application outputs the following program.

```
$data = ‘\‘; securityCheck(); $data .= ’&f=exit#’;\n f();
```

No code has been injected in this alternative output program; the injected `\` is part of a (noncode) string literal. However, because the injected symbol escapes the apostrophe that would have terminated the first string literal, the string literal continues until the next (non-escaped) apostrophe. As a result, the call to `securityCheck` is bypassed, the function `f` is updated to be

[2]We're grateful to Mike Samuel of Google for creating the first version of this example.

the `exit` function, and finally, because `#` begins a comment that continues until the line break, the `exit` function (i.e., `f`) is invoked, shutting down the web server and causing a denial of service.

4.2 Existing Solutions

Although much effort has been made to understand and prevent injection attacks, the previous work in this area has focused on *code*-injection attacks.

Conventionally, code-injection attacks are considered to occur whenever an application's input alters the intended syntactic structure of its output program. Bisht, Madhusudan, and Venkatakrishnan call this "a well-agreed principle in other works on detecting SQL injection" [15]. Indeed, this definition has appeared in many documents: [20, 93, 55, 108, 53, 11, 110, 86, 60, 18, 15, 67]. Although a few papers define CIAOs in other ways (e.g., CIAOs occur exactly when keywords or operators get injected, including apostrophes used to form string values in SQL [84, 52], or when injected strings span multiple tokens [113]), the conventional definition dominates the literature.

However, the conventional definition of code-injection attacks has inherent problems: some code-injection attacks do not alter the syntactic structures of output programs, while some non-attacks do. To illustrate these problems, Sections 4.2.1 and 4.2.2 discuss the conventional definitions of code-injection attacks used by SqlCheck [108, 110] and Candid [11, 15].

4.2.1 SqlCheck

SqlCheck considers the intended syntactic structure of an output program to be any parse tree in which each injected input is the complete derivation of one terminal or nonterminal. For example, parsing the output program `SELECT balance FROM acct WHERE password='' OR 1=1 --'` produces a parse tree in which the injected symbols `' OR 1=1 --` are not the complete sequence of leaves for a single terminal or nonterminal ancestor; SqlCheck therefore recognizes this CIAO.

However, some of what SqlCheck considers intended (i.e., non-attack) structures are actually attacks. For example, parsing the output program `SELECT balance FROM acct WHERE pin=exit()` produces a tree in which the input symbols `exit()` are the complete sequence of leaves for a single nonterminal (function-call) ancestor. Hence, SqlCheck does not recognize this CIAO as an attack. Similarly, an output program of the form `...WHERE flag=1000>GLOBAL` wouldn't be recognized as an attack, despite the injection of a greater-than operator (which may allow an attacker to efficiently extract the value of the `GLOBAL` variable, by performing a binary search over its range). Although SqlCheck allows policy engineers to specify a set of terminal and nonterminal ancestors that inputs may derive from—so engineers could disallow inputs derived as function-call and comparison expressions—it's unclear how an engineer would know exactly which ancestors to allow derivations from. Moreover, engineers may wish to sometimes allow, and sometimes disallow, inputs to derive from particular terminals and nonterminals, which is impossible in SqlCheck.

Conversely, some of what SqlCheck considers unintended (i.e., attack) structures are actually not attacks. For example, an application might input two strings, a file name `f` and a file extension `e`, and concatenate them to generate the program `SELECT * FROM properties WHERE filename='f.e'`. Although the user has injected no code, SqlCheck flags this output as a code-injection attack because the user's inputs are not complete sequences of leaves for a single terminal or nonterminal ancestor. In this case, the immediate ancestor of the user's inputs would (assuming a typical grammar) be a string literal, but neither of the user's inputs form a complete string literal—they're missing the dot and single-quote symbols.

The Candid papers describe other, lower-level problems with SqlCheck's definitions [11, 15].

4.2.2 Candid

Candid considers the intended syntactic structure of an output program, generated by running application A on input I, to be whatever syntactic structure is present in the output of A on input $VR(I)$. Here VR is a (valid representation) function that converts any input I into an input I' known to (1) be valid (i.e., non-CIAO-inducing) and (2) cause A to follow the same control-flow path as it would on input I. Candid begins by assuming this VR function exists, while acknowledging that it does not; in this basic case, Candid defines a CIAO to occur when A's output on input I has a different syntactic structure from A's output on input $VR(I)$.

Besides the nonexistence of function *VR*, there are some problems with this definition of code-injection attacks. First, the definition is circular; the attacks are defined in terms of *VR*, which itself is assumed to output non-code-injection-attack-inducing inputs (i.e., the definition of code-injection attacks relies on the definition of *VR*, which relies on the definition of code-injection attacks). Second, the definition assumes that multiple valid syntactic structures cannot exist. For example, suppose *VR*(`‘,’`)=`aaa` and application *A* on input `‘,’` outputs `SELECT * FROM t WHERE name IN (‘a’,‘b’)`, while *A* on input `aaa` executes in the same way to output `SELECT * FROM t WHERE name IN (‘aaaab’)`. Both of these outputs are valid SQL programs, yet the programs have different syntactic structures (a two-element list versus a single-element list), and neither exhibits a CIAO (in no case has *code* been injected; only values, which take no steps dynamically, have been injected). CANDID would classify the non-CIAO input of `‘,’` as an attack in this case.

To deal with the nonexistence of function *VR*, CANDID attempts to approximate *VR* by defining $VR(I)$ to be 1 when I is an integer and $a^{|I|}$ when I is a string (where $a^{|I|}$ is a sequence of a's having the same length as I). Supplying a concrete definition of *VR* resolves the circularity problem in CANDID's basic definition of CIAOs, but it doesn't resolve the second problem described in the previous paragraph (that multiple valid syntactic structures may exist).

Moreover, CANDID's approximation of *VR* creates new problems:

- The approximation incorrectly assumes a string of a's or a 1 could never be attack inputs. An application could inject an input a or 1 into an output program as part of a function call, field selection, or even keyword (e.g., `and`), all of which could be code-injection attacks. For example, suppose an application outputs a constant string, echoes its input, and then outputs parentheses; on input `exit` it outputs the program `...pin=exit()`. CANDID would not recognize this code-injection attack because the application outputs `...pin=aaaa()` on input `aaaa`, which has the same syntactic structure as the `...pin=exit()` output. The problem here is that `aaaa` is actually an attack input for this application.

- The approximation may also cause benign inputs to be detected as attacks. For example, suppose an application outputs `SELECT * FROM t WHERE flag=TRUE` on input `TRUE`, and follows the same control-flow path to output `SELECT * FROM t WHERE flag=`<u>`aaaa`</u> on input VR(`TRUE`)=`aaaa`. Because these two output programs have different syntactic structures (a boolean literal versus a variable identifier), CANDID would flag the input `TRUE` as an attack, even though the user has injected no code.

- The approximation can also break applications, as discussed in [15]. To illustrate this problem, let's consider the application `if(input<2) then restart() else output(1/(input-1))`. CANDID cannot in general operate on this application because it evaluates applications on both actual (I) and candidate ($VR(I)$) inputs, while following the control-flow path required to evaluate the actual input. In this case, whenever the application's actual input is greater than one, CANDID will try to evaluate `1/(input-1)` on the candidate input 1, which causes the application to halt with a divide-by-zero error, despite there being no errors in CANDID's absence.

It could be argued that the example applications in the bullets above would be uncommon in practice. But limiting the definition of code-injection attacks to common applications obligates us to define what makes an application common, so we can test whether a given application is "common" enough for the definition of code-injection attacks to apply. Even then, one couldn't say anything about code-injection attacks in uncommon applications.

4.2.3 CIAOs

Still other work detects code-injection attacks based on whether untrusted inputs get used as non-values (i.e., non-normal-form terms) in output programs [94]. Although we believe that this technique detects code-injection attacks precisely (i.e., lacks false positives and negatives), it is tailored to code-injection attacks and cannot detect noncode-injection attacks. Hence, false negatives arise when using the techniques of [94] to detect noncode-injection attacks. In contrast, this dissertation presents definitions and techniques for precisely detecting general injection attacks, including noncode-injection attacks.

4.2.4 Parameterized Queries

In practice, a commonly recommended technique for preventing injection attacks is to use *parameterized queries* [109], wherein applications create output programs containing placeholders, or "holes", for untrusted inputs. For example, an application might create an output program that has a single placeholder for a string literal; to output a program, the application provides a string to fill that placeholder. In this way, parameterized queries can limit injections to filling in holes for string, numeric, or other kinds of literals, thus preventing both code- and noncode-injection attacks.

While effective at preventing injection attacks, parameterized queries have significant disadvantages:

- Although parameterized queries are a standard feature of many SQL dialects, they are not supported by other common output-program languages such as HTML or bash. An output-program language must provide support for parameterized queries before an application can use them.

- It's the responsibility of application programmers to use parameterized queries. Unfortunately, many application programmers are not doing so, as evidenced by the prevalence of injection vulnerabilities [79, 85, 89].

- Once the decision to use parameterized queries is made, modifying existing applications to use them is a manual, time-consuming process. Application programmers must find all possible ways for programs to be output, and replace those programs with new versions containing the appropriate placeholders. If even one output program is not replaced, the application will remain vulnerable to injection attacks.

4.3 A Novel Technique for Preventing Injection Attacks

To address the problems with existing solutions for injection attacks, this dissertation presents a new definition of injection attacks, one that includes both code- and noncode-injection attacks. The definition of injection attacks is based on whether untrusted inputs affect output programs in any way besides inserting or expanding noncode tokens (such as string, integer, or float literals). When untrusted inputs only insert or expand noncode tokens in an output program, we say that the output program satisfies the *NIE property* (Noncode Insertion or Expansion). On the other hand, if inputs affect an output program beyond inserting or expanding noncode tokens, we say that the output program exhibits a *BroNIE* (Broken NIE).

The NIE property restricts untrusted inputs in ways that are similar to parameterized queries; both techniques require untrusted inputs to fill noncode-token "holes" in output programs. With parameterized queries, the holes for untrusted-input tokens are manually specified by application programmers; with this dissertation's techniques, all holes for untrusted-input tokens are automatically confined to being noncode (e.g., a string or numeric literal). Because this dissertation's techniques are widely applicable and require no modifications to existing applications, the techniques avoid the disadvantages of parameterized queries. However, this dissertation's techniques rely on runtime monitoring for injection-attack detection, implying higher runtime overhead than parameterized queries.

This section formalizes criteria for determining when a BroNIE has occurred.

4.3.1 Notation and Assumptions

An application vulnerable to BroNIEs outputs programs in some language L (e.g., SQL) that has a finite concrete-syntax alphabet Σ_L (e.g., the set of printable Unicode characters). These output programs, which we also call L-programs, are finite sequences of Σ_L symbols that each form an element of L. For L-program $p = \sigma_1\sigma_2..\sigma_n$, let $|p| = n$ and $p[i] = \sigma_i$. For a sequence S, the replacement of item t with item t' (i.e., the substitution of t' for t in S) is denoted $[t'/t]S$.

This paper makes a few assumptions about output-program languages. All output-program languages under consideration have well-defined functions for:

- Computing the free variables of program terms. Terms are called *open* if they contain free variables (e.g., `1+x`), and are otherwise *closed* (e.g., `1+2`).

- Testing whether program terms are *values*. Values are the "fully evaluated" terms of a programming language, such as literals, pointers, objects, lists and tuples of other values, lambda terms, etc.

- Tokenizing output programs. Function $\mathit{tokenize}_L(\sigma_1..\sigma_n)$ returns the sequence of tokens within the string $\sigma_1..\sigma_n$ (assuming $\sigma_1..\sigma_n$ is lexically valid; otherwise, $\mathit{tokenize}_L(\sigma_1..\sigma_n)$ returns the empty sequence). A token of *kind* τ composed of symbols $\sigma_i..\sigma_j$ is represented as $\tau_i(\sigma_i..\sigma_j)_j$. For example, for program $q =$ `SELECT * FROM orders WHERE s='' OR i<3`, $\mathit{tokenize}_{SQL}(q)$ returns tokens $SELECT_1(\texttt{SELECT})_6$, $STAR_8(\texttt{*})_8$, $FROM_{10}(\texttt{FROM})_{13}$, $ID_{15}(\texttt{orders})_{20}$, $WHERE_{22}(\texttt{WHERE})_{26}$, $ID_{28}(\texttt{s})_{28}$, $EQUALS_{29}(\texttt{=})_{29}$, $STRING_{30}(\texttt{''})_{31}$, $OR_{33}(\texttt{OR})_{34}$, $ID_{36}(\texttt{i})_{36}$, $LESS_{37}(\texttt{<})_{37}$, and $INT_{38}(\texttt{3})_{38}$. Predicate $TR_L(p, i)$ (tokenizer-removed) holds iff i is not within the bounds of any token in $\mathit{tokenize}_L(p)$. For example, $TR_{SQL}(q, i)$ holds for all $i \in \{7, 9, 14, 21, 27, 32, 35\}$.

This dissertation omits the L subscript from the $\mathit{tokenize}_L$ and TR_L functions when the output-program language is clear from context. Also, because all tokens are labeled with begin and end indices, it's trivial to convert a set of nonoverlapping tokens into the equivalent sequence of tokens (and vice versa). This paper therefore treats sequences of tokens as sets, and vice versa, as convenient.

4.3.2 Defining Injection

Applications vulnerable to BroNIEs output programs based on inputs from trusted and untrusted sources. *Injected* symbols are those that originate from untrusted sources and propagate unmodified through an application into its output program. A BroNIE occurs when injected symbols affect output programs in any way besides inserting or expanding noncode tokens.

We rely on the well-studied concept of taint tracking [52, 83, 113, 93, 105] to determine which output-program symbols originate from untrusted sources and are therefore injected. At a high level, a *taint-tracking application* works by replacing all symbols input from untrusted sources with their tainted versions and preserving these taint metadata during all copy and output operations. Provided that the application only taints symbols when they are being input from an untrusted source, and never untaints symbols, the injected symbols in the output program are exactly those that are tainted. As we have been underlining injected symbols, we use the same notation to mark tainted symbols.

Definition 16. *For all alphabets* Σ*, the* tainted-symbol alphabet $\underline{\Sigma}$ *is* $\{\sigma \mid \sigma \in \Sigma \vee (\exists \sigma' \in \Sigma : \sigma = \underline{\sigma'})\}$*.*

Definition 17. *For all languages* L *with alphabet* Σ*, the* tainted output language $\underline{L}$ *with alphabet* $\underline{\Sigma}$ *is* $\{\sigma_1..\sigma_n \mid \exists\sigma'_1..\sigma'_n{\in}L\text{: } \forall i{\in}\{1..n\}\text{: } (\sigma_i{=}\sigma'_i \vee \sigma_i{=}\underline{\sigma'_i})\}$*.*

Definition 18. *For all alphabets* Σ *and symbols* $\sigma \in \underline{\Sigma}$*, the predicate injected*(σ) *is true iff* $\sigma \notin \Sigma$*.*

For example, taint-tracking application `output('Hello' + input() + '!')` on untrusted input `World` will output the program `Hello` $\underline{\texttt{World}}$`!` because the initial input is replaced by its tainted version $\underline{\texttt{World}}$, and the taint metadata are preserved by the string-concatenation and output operations.

Because taint tracking is a well-studied technique, the remainder of this dissertation assumes that applications can track taints and thus output programs in which the injected (i.e., tainted) symbols are underlined.

4.3.3 Defining Noncode

Intuitively, noncode symbols in output programs are those that are dynamically passive; they specify no computation to be performed during execution. Code symbols, on the other hand, are dynamically active; they specify computation that could be performed during execution.

This paper considers a program's noncode symbols to be exactly those that are either (1) removed by the tokenizer or (2) within a closed value:

1. Although previous work sometimes allowed tokenizer-removed symbols such as whitespace or comments to be code [94], we believe that, because tokenizer-removed symbols are dynamically passive and cannot specify computation, it is more intuitive and accurate to consider such symbols noncode.

2. Closed values are operationally irreducible and thus specify no dynamic computation. Typical values include literals, pointers, objects, and tuples of other values. Open values are excluded because they specify the dynamic computation of substituting a term for a free variable during execution.

When $p[i]$ is noncode (where p is an output program), we write $Noncode(p, i)$. Otherwise, we write $Code(p, i)$.

Definition 19. *For all L-programs $p = \sigma_1..\sigma_n$ and position numbers $i \in \{1..|p|\}$, predicate $Noncode(p, i)$ holds iff $TR_L(p, i)$ or there exist low and high symbol-position numbers $l \in \{1..i\}$, $h \in \{i..|p|\}$ such that $\sigma_l..\sigma_h$ is a closed value in p.*

Tokens composed entirely of noncode symbols are called *noncode tokens*. The set of all noncode tokens in a program p is $noncodeToks(p)$.

4.3.4 An Aside: Defining *Code*-injection Attacks

Using the predicates for determining which symbols are injected (Definition 18), and which are noncode (Definition 19), we can define Code-Injection Attacks on Output programs (CIAOs) as occurring exactly when an output program contains an injected code symbol.

Definition 20. *A* CIAO *occurs exactly when a taint-tracking application outputs $\underline{L}$-program $p = \sigma_1..\sigma_n$ such that $\exists i \in \{1..n\} : (injected(\sigma_i) \ \wedge \ Code(p, i))$.*

4.3.5 Defining BroNIEs

Because BroNIEs occur when injected symbols affect output programs beyond inserting or expanding noncode tokens, they can be detected by observing how a program's sequence of tokens is affected by the removal of its injected symbols. Intuitively, removing all injected symbols from the output program should only affect the sequence of tokens in the following ways:

1. Some noncode tokens may no longer be present.

2. Some noncode tokens may become smaller but should not change kind (e.g., string literals should not become integer literals).

To formalize this intuition, we need to consider the sequence of tokens obtained by removing all injected symbols from an output program. The injected symbols cannot simply be deleted; doing so would affect the indices of tokens that follow the injected symbols. Instead, each injected symbol is replaced with an ε (the empty string). The sole purpose of an ε symbol is to hold the place of an injected symbol; ε's are otherwise ignored. Because the resulting string contains only uninjected symbols, it can be considered a *template* of the program.

Definition 21. *The* template *of a program p, denoted $[\varepsilon/\underline{\sigma}]p$, is obtained by replacing each injected symbol in p with an ε.*

For example, let program $r = \underline{\mathtt{1}}\mathtt{2}\underline{\mathtt{3}}\mathtt{+1}$. Then $[\varepsilon/\underline{\sigma}]r$ is $\varepsilon\mathtt{2}\varepsilon\mathtt{+1}$ (which is equivalent to `2+1`) and contains the tokens $INT_2(\mathtt{2})_2$, $PLUS_4(\mathtt{+})_4$, and $INT_5(\mathtt{1})_5$.

The definition of BroNIEs also relies on notions of token insertion and expansion. Token insertion is straightforward, but we need to be clear about the meaning of token expansion: injected symbols may expand noncode tokens by increasing their ranges of indices and corresponding strings of program symbols.

Definition 22. *A token $t = \tau_i(v)_j$ can be* expanded *into token $t' = \tau'_{i'}(v')_{j'}$, denoted $t \preceq t'$, iff (1) $\tau{=}\tau'$, (2) $i' \leq i \leq j \leq j'$, and (3) v is a subsequence of v'.*

Whether the $\preceq$ operator refers to execution prefixing or token expansion will always be clear from context.

Returning to the example above, token $INT_2(\mathtt{2})_2$ in $[\varepsilon/\underline{\sigma}]r$ can be expanded into token $INT_1(\underline{\mathtt{1}}\mathtt{2}\underline{\mathtt{3}})_3$ in r. That is, $INT_2(\mathtt{2})_2 \preceq INT_1(\underline{\mathtt{1}}\mathtt{2}\underline{\mathtt{3}})_3$.

We can now formally specify when an output program exhibits only noncode insertion or expansion (NIE). Given a program p and its template $[\varepsilon/\underline{\sigma}]p$, it should be possible to get to the sequence of tokens in p from the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ by only inserting or expanding noncode tokens. If the sequence of tokens in p can be reached from the sequence of tokens $[\varepsilon/\underline{\sigma}]p$ in this way, we say that p satisfies the NIE property; otherwise it exhibits a BroNIE.

Definition 23. *An L-program p* satisfies the NIE property *iff there exist:*

- $I \subseteq noncodeToks(p)$ *(i.e., a set of p's inserted noncode tokens),*
- $n \in \mathbb{N}$ *(i.e., a number of p's expanded noncode tokens),*
- $\{t_1..t_n\} \subseteq tokenize([\varepsilon/\underline{\sigma}]p)$ *(i.e., a set of template tokens to be expanded), and*
- $\{t'_1..t'_n\} \subseteq noncodeToks(p)$ *(i.e., a set of p's expanded noncode tokens)*

such that:

- $t_1 \preceq t'_1,\ \ldots, t_n \preceq t'_n$*, and*
- $tokenize(p) = ([t'_1/t_1]..[t'_n/t_n]tokenize([\varepsilon/\underline{\sigma}]p)) \cup I$.

Definition 24. *A* BroNIE *(Broken NIE) occurs exactly when a taint-tracking application outputs a program that violates the NIE property.*

4.4 Demonstrations of the BroNIE Definition

Let us consider several examples of how Definition 24 classifies programs as either attacks or non-attacks. Although all but one of this section's examples are presented in SQL, the underlying concepts apply to other languages as well.

4.4.1 Example 1

Consider the following simple output program.

$$\texttt{SELECT * FROM files WHERE numEdits > }\underline{\texttt{0}}\texttt{ AND name='}\underline{\texttt{file}}\texttt{.}\underline{\texttt{ext}}\texttt{'}$$

This query returns files named `file.ext` that have been edited at least once. Figure 4.1 shows how this program and its template are tokenized. This program does not exhibit a CIAO; all injected symbols are part of integer or string values. Neither is it a BroNIE; the injected symbols only cause noncode insertion and expansion, as depicted in Figure 4.1. More formally, the sequence of tokens in the template can be made into the sequence of tokens in the program by inserting the noncode token $INT_{38}(\texttt{0})_{38}$ and expanding the token $STRING_{49}(\texttt{'.'})_{58}$ into the (noncode) token $STRING_{49}(\texttt{'}\underline{\texttt{file}}\texttt{.}\underline{\texttt{ext}}\texttt{'})_{58}$. Hence, the definition of BroNIEs matches our intuition that this program does not exhibit an attack.

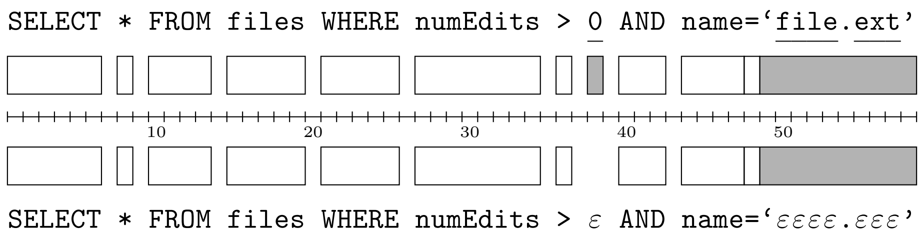

SELECT * FROM files WHERE numEdits > 0 AND name='file.ext'
10
20
30
40
50
SELECT * FROM files WHERE numEdits > ε AND name='εεεε.εεε'

Figure 4.1. A depiction of how the Example-1 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded.

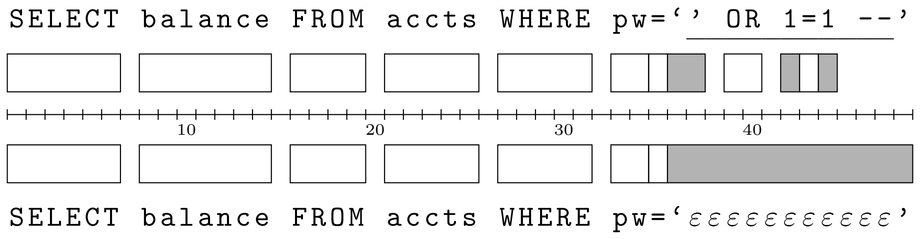

SELECT balance FROM accts WHERE pw='' OR 1=1 --'
10
20
30
40
SELECT balance FROM accts WHERE pw='εεεεεεεεεεε'

Figure 4.2. A depiction of how the Example-2 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded.

4.4.2 Example 2

Turning our attention to programs that do exhibit injection attacks, let's return to the first malicious output program presented in Section 4.1.

```
SELECT balance FROM accts WHERE pw='' OR 1=1 --'
```

This query returns all balances from the `accts` table because the `WHERE` clause is a tautology. The application that output this program tries to access only balances for which a password is known, but the malicious input circumvents the password check.

Figure 4.2 shows how this program and its template are tokenized. The program exhibits a CIAO and a BroNIE: a CIAO because the injected `O`, `R`, and `=` symbols are code, and a BroNIE because the injected symbols insert code tokens and *contract* the $STRING_{36}(\texttt{''})_{48}$ token into $STRING_{36}(\texttt{''})_{37}$.

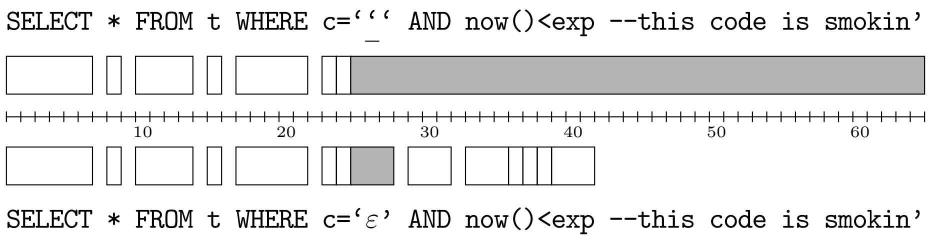

SELECT * FROM t WHERE c=‘‘‘ AND now()<exp --this code is smokin’
10
20
30
40
50
60
SELECT * FROM t WHERE c=‘ε’ AND now()<exp --this code is smokin’

Figure 4.3. A depiction of how the Example-3 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded.

4.4.3 Example 3

Because injections that cause CIAOs must also cause BroNIEs (as will be shown in Theorem 7), the remainder of this section focuses on non-CIAO examples of BroNIEs, such as the following.

```
SELECT * FROM t WHERE c=‘‘‘ AND now()<exp --this code is smokin’
```

The application that outputs this program attempts to find all unexpunged records with a given column value. For instance, the application could be querying a table of juvenile crimes (such as truancy) that can legally only be displayed when the offenders are not yet adults. However, the application appends a seemingly harmless, boastful comment to all output queries that allows an injected apostrophe to circumvent the expungement check by escaping the string literal's terminator (in SQL, apostrophes within string literals are escaped by a second apostrophe). If the attacker has control over the column being queried (e.g., it could be a comment column), then he or she can illegally access records after they have been expunged.

Figure 4.3 presents the tokenizations of this example program and its template. This program does not exhibit a CIAO; the injected symbol is part of a noncode (string) token. On the other hand, this program does exhibit a BroNIE; the injected symbol violates the NIE property by deleting code tokens.

4.4.4 Example 4

The following output program also contains a non-CIAO BroNIE.

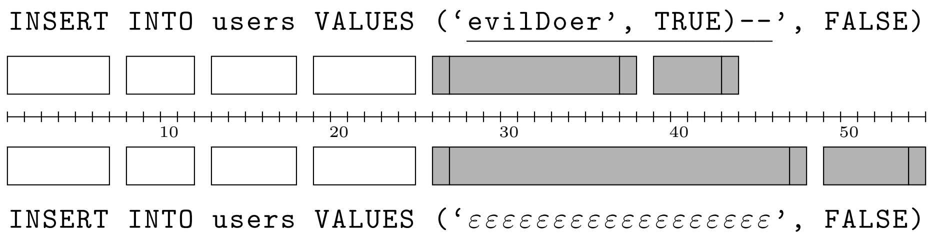

INSERT INTO users VALUES (‘evilDoer’, TRUE)--’, FALSE)
10
20
30
40
50
INSERT INTO users VALUES (‘εεεεεεεεεεεεεεεεεε’, FALSE)

Figure 4.4. A depiction of how the Example-4 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded. The tokens for commas and parentheses are considered noncode because they are within a tuple value; tuple terms are values when all subterms are values.

$$\texttt{INSERT INTO users VALUES (`}\underline{\texttt{evilDoer', TRUE)--}}\texttt{', FALSE)}$$

This program creates a new user by inserting a record into the `users` table. Each record contains a field for the username and a boolean flag indicating whether the user has administrator privileges. By hardcoding a `FALSE` value for the second element of new-user records, the application that output this program attempts to ensure that all accounts it creates do not have administrator privileges. However, in this case an attacker has supplied a malicious username that causes the application to create an administrator-privileged account.

Figure 4.4 shows how this program and its template are tokenized. None of the injected symbols in the program are code symbols, so it does not exhibit a CIAO. However, it does exhibit a BroNIE; the injected symbols cause noncode contraction and deletion. For example, the last three noncode tokens in the template are not present in the output program. Thus, Definition 24 correctly considers this output program to be an attack.

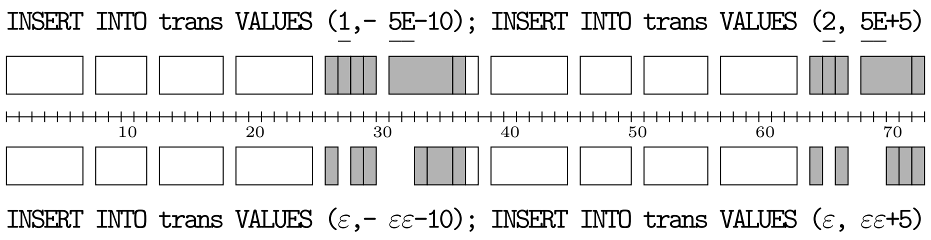
4.4.5 Example 5

The following output program will serve as a final example of a non-CIAO BroNIE in SQL.

$$\texttt{INSERT INTO trans VALUES (}\underline{\texttt{1}}\texttt{,- }\underline{\texttt{5E}}\texttt{-10);}$$

$$\texttt{INSERT INTO trans VALUES (}\underline{\texttt{2}}\texttt{, }\underline{\texttt{5E}}\texttt{+5)}$$

Here, an intern-authored application outputs programs to handle money transfers from one account (e.g., account number 1) to another (e.g., account number 2). The application applies a service



INSERT INTO trans VALUES (1,- 5E-10); INSERT INTO trans VALUES (2, 5E+5)
10
20
30
40
50
60
70
INSERT INTO trans VALUES (ε,- εε-10); INSERT INTO trans VALUES (ε, εε+5)

Figure 4.5. A depiction of how the Example-5 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded. The tokens for commas and parentheses are considered noncode because they are within a tuple value; tuple terms are values when all subterms are values.

charge of $10 to the paying account, but is currently running a special that transfers an extra $5 into the receiving account. However, by appending an 'E' to the money transfer amount, a malicious user can drastically affect the transfer process—in the above program, the paying account is only charged $0.0000000005 while the receiving account is credited $500,000.

Figure 4.5 shows that this output program exhibits a non-CIAO BroNIE. The output program does not exhibit a CIAO because no code has been injected; only components of float literals have been injected. On the other hand, the output program does exhibit a BroNIE because the injections delete the $MINUS_{33}(\texttt{-})_{33}$ and $PLUS_{70}(\texttt{+})_{70}$ tokens and transform two INT tokens into $FLOAT$ tokens (recall that Definition 22 does not allow tokens of one kind to expand into tokens of another kind).

4.4.6 Example 6

To demonstrate the effectiveness of these techniques in other languages, we return to the following output program from Section 4.1.

```
$data = '\'; securityCheck(); $data .= '&f=exit#';\n f();
```

Because apostrophes within string literals are escaped by backslashes in this language, the injected symbol bypasses the call to the `securityCheck` function, garbles the contents of the `data` variable, and causes the `exit` function to be invoked instead of the `f` function.

The tokenizations of this program and its template are depicted in Figure 4.6. The program does not exhibit a CIAO; the only injected symbol is part of a string literal (i.e., a noncode value).

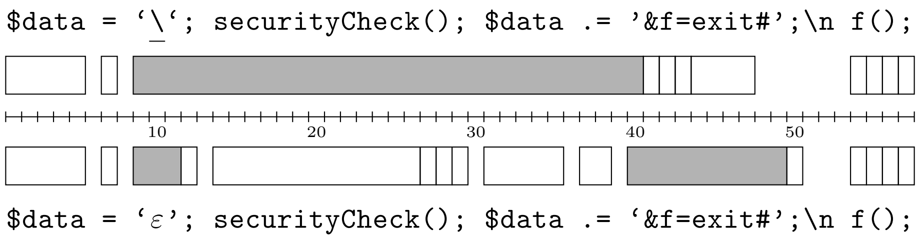

$data = '\'; securityCheck(); $data .= '&f=exit#';\n f();
10
20
30
40
50
$data = 'ε'; securityCheck(); $data .= '&f=exit#';\n f();

Figure 4.6. A depiction of how the Example-6 program and its template are tokenized. The program is on the top, the template on the bottom, and noncode tokens are shaded.

However, the program does exhibit a BroNIE; the injected symbol deletes code and noncode tokens, and inserts code tokens, thus violating the NIE property.

4.5 Analysis of the BroNIE Definition

This section explores implications of the definitions in Section 4.3.

4.5.1 Relationship between, and Prevalence of, CIAOs and BroNIEs

Every application vulnerable to CIAOs is also vulnerable to BroNIEs.

Theorem 7. *If a program exhibits a CIAO, then it exhibits a BroNIE.*

Proof. Let p be an output program that exhibits a CIAO. Then at least one of p's code symbols is injected; let σ_c be one of these injected code symbols and t be the token containing σ_c. Program p satisfies the NIE property iff the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ can be made equal to the sequence of tokens in p, subject to the constraints of Definition 23. Observe that t cannot be in $[\varepsilon/\underline{\sigma}]p$; no token in $[\varepsilon/\underline{\sigma}]p$ can have the same text as t unless its begin and/or end indices are different because, in $[\varepsilon/\underline{\sigma}]p$, σ_c will be replaced by an ε. Because t is a code token (it contains the code symbol σ_c), Definition 23 does not allow t to be inserted into the token sequence of $[\varepsilon/\underline{\sigma}]p$, nor for a token in $[\varepsilon/\underline{\sigma}]p$ to be expanded into t. Hence, the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ cannot be made equal to the sequence of tokens in p by only inserting and/or expanding noncode tokens, so p exhibits a BroNIE. □

Theorem 8 provides a formal basis for the widely-accepted rule of thumb that it's unsafe to use unvalidated input during query construction [109, 102]. It states that if an application always includes an untrusted input (i_m) verbatim in its output (without even inspecting the input), and the same application has some input ($v_1, .., v_n$) for which it outputs a valid SQL program, then there exists a way to construct an attack input (a_m) such that the application's output will exhibit a CIAO and therefore a BroNIE. Theorem 8 is a generalization of Theorem 9 in [94], which was limited to CIAOs in an idealized subset of SQL called SQL Diminished; in contrast, Theorem 8 below applies to full SQL.

Theorem 8. *For all n-ary functions A and (n-1)-ary functions A' and A'', if $\forall i_1, .., i_n\colon A(i_1, .., i_n) = A'(i_1, .., i_{m-1}, i_{m+1}, .., i_n)\underline{i_m}A''(i_1, .., i_{m-1}, i_{m+1}, .., i_n)$, where $1 \le m \le n$, and $\exists v_1, .., v_n\colon (v_m \in \Sigma^+_{SQL} \wedge A(v_1, .., v_n) \in SQL)$, then $\exists a_1, .., a_n\colon A(a_1, .., a_n) \in SQL$ and $A(a_1, .., a_n)$ exhibits a CIAO and a BroNIE.*

Proof. Observe that changing v_m to any a_m, without changing any of the other inputs to A, will cause A to output the same program but with a_m instead of v_m, because A' and A'' are independent of i_m. Now construct a_m as follows:

$$a_m = v_m A''(v_1, .., v_{m-1}, v_{m+1}, .., v_n)\backslash n;\ \textit{drop table } t;\ A'(v_1, .., v_{m-1}, v_{m+1}, .., v_n)v_m$$

. This construction causes $A(v_1, .., v_{m-1}, a_m, v_{m+1}, .., v_n)$ to output the string $p = A(v_1, .., v_n)\underline{\backslash n;\ \textit{drop table } t;}A(v_1, .., v_n)$. Because $A(v_1, .., v_n) \in SQL$, so too is output-program p. By Definition 20, p exhibits a CIAO due to the injected `DROP` statement, as well as any other code symbols in a_m. By Theorem 7, p also exhibits a BroNIE. □

It is also straightforward to prove, using the same techniques from [94], that neither static nor black-box mechanisms can precisely prevent BroNIEs. That is, precise detection of BroNIEs requires dynamic, white-box mechanisms.

4.5.2 An Algorithm for Precisely Detecting BroNIEs

Given that applications commonly fail to validate untrusted inputs [102], it would be beneficial to have mechanisms for automatically preventing injection attacks. At a high level, BroNIEs can be precisely and automatically prevented by:

- instrumenting the target application with a taint-tracking mechanism,
- interposing between the target application and the environment that evaluates the application's output programs,
- detecting whether output programs satisfy the NIE property, and
- only executing programs that do not exhibit BroNIEs.

Algorithm 4.7 is a psuedocode implementation of this mechanism. The algorithm detects BroNIEs by iterating through the sequences of tokens in the output program and its template while ensuring that the two token streams can be made equal subject to the constraints of Definition 23 (i.e., by only inserting noncode tokens into, or expanding noncode tokens within, the program's template).

Theorem 9. *Algorithm 4.7 executes program p iff p doesn't exhibit a BroNIE.*

Proof. We prove the if direction; the only-if direction is similar. By assumption, p satisfies the NIE property. Hence, the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ can be made equal to the sequence of tokens in p by expanding tokens in $[\varepsilon/\underline{\sigma}]p$ into noncode tokens in p, and/or inserting noncode tokens in p into $[\varepsilon/\underline{\sigma}]p$. Variables i and j are initialized to 1. For every token present in p and $[\varepsilon/\underline{\sigma}]p$, both i and j will be incremented by the first branch of the `if` statement (Line 7). For every token in $[\varepsilon/\underline{\sigma}]p$ that needs to be expanded into a noncode token in p, i and j will be incremented by the second branch of the `if` statement (Lines 8–9). For every noncode token in p that needs to be inserted into $[\varepsilon/\underline{\sigma}]p$, i will be incremented by either the third branch of the `if` statement (Line 10) or the second `while` loop (Line 13). After both `while` loops have completed, both i and j will be greater than the lengths of their token sequences, so p will be executed on Line 14. $\square$

```
Input: Taint-tracking application $A$ and inputs $T$, $U$ (trusted, untrusted)
Ensure: $A$'s output-program $p$ is executed iff it doesn't exhibit a BroNIE
 1: $p \leftarrow A$ ($T$, Taint($U$))
 2: $pgmTokens \leftarrow$ tokenize($p$)
 3: $temTokens \leftarrow$ tokenize($[\varepsilon/\underline{\sigma}]p$)
 4: MarkNoncodeToks($pgmTokens$)
 5: $i \leftarrow j \leftarrow 1$
 6: while $i \leq pgmTokens$.length and $j \leq temTokens$.length:
 7:    if $pgmTokens[i] = temTokens[j]$ then increment $i$ and $j$
 8:    else if $pgmTokens[i]$.isNoncode and $temTokens[j] \preceq pgmTokens[i]$ then
 9:       increment $i$ and $j$
10:    else if $pgmTokens[i]$.isNoncode then increment $i$
11:    else throw $BronieException$
12: // Handle any trailing noncode tokens in the program
13: while $i \leq pgmTokens$.length and $pgmTokens[i]$.isNoncode: increment $i$
14: if $i > pgmTokens$.length and $j > temTokens$.length then Execute($p$)
15: else throw $BronieException$
```

Figure 4.7. A BroNIE-preventing mechanism.

Theorem 10. *The BroNIE-detection part of Algorithm 4.7 (i.e., Lines 2–13) executes in $O(n)$ time, where n is the length of the output program.*

Proof. Each call to *tokenize* executes in $O(n)$ time and produces $O(n)$ tokens. After the program tokens are marked as either code or noncode (taking $O(n)$ time), the two `while` loops execute; each iteration takes constant time, and each loop executes $O(n)$ times. Afterward, if a BroNIE has not yet been detected, deciding whether a BroNIE has occurred takes constant time. Thus, Algorithm 4.7 runs in $O(n)$ time. □

4.6 Preventing BroNIEs with Gray Policies and Silhouette Judges

With an improved understanding of which output programs exhibit injection attacks, we now focus on writing gray policies that use the BroNIE definition.

We begin by defining the event set E. Events of interest include:

- `emit(p)`: these events occur when applications output a program p.
- `exec(p)`: these events occur when the runtime monitor allows a program p to be executed.

- `deny(p)`: these events occur when the runtime monitor prevents a program p from being executed.

While applications may be capable of performing many other actions, these are the only events that a runtime monitor dedicated to preventing injection attacks will interact with.

With the event set defined, gray properties and policies for preventing injection attacks can be defined. Most straightforwardly, a property could reduce the security of a trace for each prefix it has that ends with a BroNIE being executed. To handle the special case of infinite traces (which may have an infinite number of such prefixes), the gray property is constructed to compute the security value of each of the trace's prefixes, and mapping the trace to the greatest lower bound of these security values.

$$g_1(x) = \inf \left\{ \frac{1}{1 + |\{x''; \langle e, \texttt{exec(p)} \rangle \mid x''; \langle e, \texttt{exec(p)} \rangle \preceq x' \wedge \mathit{BroNIE}(p)\}|} \mid x' \preceq x \right\}$$

As expected, if a trace x never executes a BroNIE, then $g_1(x) = 1$, and for every exchange in x that does execute a BroNIE, $g_1(x)$ decreases. Furthermore, every reduction in security occurs at some finite point along the trace x. Thus, g_1 is a gray safety property.

A similar construction punishes executions that deny non-BroNIE queries:

$$g_2(x) = \inf \left\{ \frac{1}{1 + |\{x''; \langle \texttt{emit(p)}, \texttt{deny(p)} \rangle \mid x''; \langle \texttt{emit(p)}, \texttt{deny(p)} \rangle \preceq x' \wedge \neg \mathit{BroNIE}(p)\}|} \mid x' \preceq x \right\}$$

Again, this is a safety property; once an execution's security has decreased, it cannot rise again.

One of the strengths of gray policies is that they provide many interesting options for composing with each other. For example, the following gray policy combines g_1 and g_2 to punish executions for both executing BroNIEs and not executing non-BroNIEs.

$$g_3(x) = w(g_1(x)) + (1 - w)(g_2(x))$$

Here, $w \in \mathbb{R}_{[0,1]}$ represents the weight given to g_1; when $w = 0.5$, g_1 and g_2 are weighted equally, while higher (lower) values of w over- (under-) emphasize g_1 compared to g_2. Other options for combining g_1 and g_2 into a single property include using the lower of g_1 and g_2, or by multiplying their security values.

$$g_4(x) = \min(g_1(x), g_2(x))$$

$$g_5(x) = g_1(x) * g_2(x)$$

Because g_1 and g_2 are both safety properties, so too are g_3, g_4, and g_5. Of these, g_4 is the most permissive, because it allows one property to misbehave "for free" whenever the other property misbehaves. For example, if an application executes a BroNIE, then the application can then deny a non-BroNIE without reducing its security.

The quantified nature of gray policies allows interesting ways of using the BroNIE definition that do not rely on a $BroNIE$ predicate. For example, Definition 23 forbids any difference in the sequences of tokens in a program and its template; instead, a gray policy might count how many "forbidden" edits are required to make the two sequences equal. To formalize this idea, let $NIE^{\circ}(p)$ be 1 when p satisfies Definition 23, and be decimated for every "forbidden" edit needed. Using this notion, a gray property can be constructed that measures the *severity* of a trace's output program executions.

$$g_6(x) = \inf\{NIE^{\circ}(p) \mid \langle e, exec(p)\rangle \in x\}$$

This gray property is also gray safety, because once a trace includes an unsafe execution, its security can never rise. Unfortunately, even programs that only require a single "forbidden" edit can be exceptionally malicious. As such, g_6 is not recommended to prevent injection attacks; it is shown here because a similar idea might be useful in other contexts.

Non-safety properties for preventing injection attacks can also be defined. For example, an execution can be evaluated based on what ratio of its executed output programs are BroNIEs.

$$g_7(x) = \begin{cases} \dfrac{|\{x''; \langle e, \texttt{exec(p)}\rangle \mid x''; \langle e, \texttt{exec(p)}\rangle \preceq x' \land \neg BroNIE(p)\}|}{|\{x''; \langle e, \texttt{exec(p)}\rangle \mid x''; \langle e, \texttt{exec(p)}\rangle \preceq x'\}|} & \text{if } x \in \mathcal{E}^* \\ 1 & \text{otherwise} \end{cases}$$

This property is a liveness property because any finite execution can increase its security by executing programs that are not BroNIEs.

Furthermore, non-property policies for injection attacks can be defined. Using the technique presented in Section 3.2, the following policy measures an application's capability of detecting injection attacks.

$$\begin{aligned} G_7(X) \;\;=\;\; & \mathit{prob}(x \text{ executes a BroNIE} \mid x \in X) \, * \\ & \mathit{prob}(x' \in X \mid x' \in \mathcal{E}^{\infty} \wedge x' \text{ does not execute a BroNIE}) \end{aligned}$$

This property is gray hyperliveness because any terminating target system can be made more secure by extending it to include infinite-length executions that do not execute BroNIEs.

With a healthy collection of gray policies, we move on to defining silhouette judges. Recall that gray policies are responsible for determining how secure target systems are, while silhouette judges are responsible for encoding users' security requirements. Thus, to define useful silhouette judges, we must first consider the security requirements of users.

First, suppose a simple security requirement that no BroNIEs are allowed to execute. This requirement can be encoded into a silhouette judge by requiring absolute security.

$$j_1(s) = s(\infty) = 1$$

Combining j_1 with the gray policy g_1 (which only considers as absolutely secure traces that do not execute BroNIEs) creates a black-and-white policy that forbids BroNIEs from executing. Similarly, combining j_1 with g_2 creates a black-and-white policy that ensures non-BroNIEs are executed when they are emitted.

This security requirement can be relaxed to instead ensure that security never dips below some threshold value t.

$$j_2(s) = \forall x \in \mathit{dom}(s) : s(x) \geq t$$

For example, combining j_2 with g_5 will create a black-and-white policy that only allows some number of mistakes from the output program (either executed BroNIEs or denied non-BroNIEs) before disallowing further exchanges, depending on the values of t in j_2 and w in g_5.

Other approaches could be used for allowing less secure traces. For example, the following silhouette judge makes sure that no part of the silhouette is overly vulnerable; it uses a sliding window of length n, and ensures that the average security inside of the window never dips below a certain threshold $t \in \mathbb{R}_{[0,1]}$.

$$j_3(s) = \forall i \in \mathbb{N} : i + n \in dom(s) \Rightarrow \sum_{j=i}^{i+n} \left(\frac{s(i)}{n} \right) > t$$

Combining silhouette j_3 with the gray policy g_7 creates a gray property that allows executions to occasionally execute BroNIEs, but if too many BroNIEs are executed in a short period of time, the silhouette judge will intervene.

Along the same lines, a silhouette judge could only allow executions to stray below a security threshold for short periods of time before returning to an acceptable security value.

$$j_4(s) = \forall i \in \mathbb{N} : (i + n \in dom(s) \wedge s(i) < t) \Rightarrow (\exists j \in \{1..n\} : s(i + j) \geq t)$$

Silhouette judges for target systems by necessity are more complex, but allow for interesting requirements. For example, a user might want to ensure that a target system become more secure as it executes. However, there is no way to know which execution will occur at runtime, so the user requires that all 2-length prefixes be more secure than all 1-length prefixes, and that all 3-length prefixes are more secure than all 2-length prefixes, and so on.

$$J_5(S) = \forall (r_1, r_2) \in \{(0.1919.., 0.2929..), (0.2929.., 0.3939..), ...\} : S(r_1) < S(r_2)$$

Combining J_5 with G_7 creates a black-and-white policy that only allows target-systems that are more precise in detecting BroNIEs as traces proceed. Such a policy would be useful for determining the effectiveness of adaptive systems for injection-attack prevention.

CHAPTER 5

CONCLUSIONS

This dissertation has presented a new model of formal security policies that allows specification of how secure target systems are, as opposed to the existing models which specify whether systems are secure. The gray model is a proper generalization of the existing models, and as demonstrated in Chapter 2, preserves several interesting qualities:

1. The only gray property that is both gray safety and gray liveness is the property that maps every trace to the highest security value (Theorem 1).
2. Similarly, the only gray policy that is both gray hypersafety and gray hyperliveness is the property that maps every target system to the highest security value (Theorem 2).
3. Every gray property can be expressed as the minimum of a single gray safety property and a single gray liveness property (Theorem 3).
4. Similarly, every gray policy can be expressed as the minimum of a single gray hypersafety property and a single gray hyperliveness property (Theorem 4).

As an added benefit, the gray model of policies can serve as a unifying framework for expressing security metrics. In Chapter 3, techniques for constructing gray policies out of existing security metrics were explored. Security metrics from varied fields, including access control, confidentiality, privacy, and network security, were used to construct a large variety of gray policies. Additionally, methods for usefully converting existing black-and-white policies to gray policies were explored.

To further demonstrate the gray model, Chapter 4 delved into the topic of injection attacks. Novel injection attacks which do not involve injecting code were explored, and problems in existing solutions were discussed. The primary contributions include:

1. A definition of injection attack that avoids the problems of existing solutions (Definition 24).
2. A proof that this new definition correctly includes code-injection attacks (Theorem 7).
3. A proof that any application which blindly copies its untrusted inputs into its outputs will be vulnerable to injection attacks (Theorem 8).
4. A linear-time algorithm for detecting and preventing injection attacks, with proofs of correctness and running time (Figure 4.7).
5. An exploration in Section 4.6 of how to model and prevent injection attacks using gray policies and silhouette judges.

5.1 Future Work

Several directions exist for future work.

One would be to design and evaluate programming languages, or other tools, for specifying gray policies and silhouette judges. As a part of this direction, it would be interesting to consider case studies, to learn which sorts of gray policies and silhouette judges seem to be more common, or practically useful.

Another direction would investigate generalizations of existing program-verification techniques, to transition from determining whether programs obey black-and-white policies to determining how well programs obey gray policies.

It would also be interesting to consider ways in which the gray security model could benefit from results known in the area of fuzzy set theory. Intuitively, gray policies are to black-and-white policies what fuzzy sets are to sets: A fuzzy set is an ordered pair (U, m), where U is a set and $m : U \to \mathbb{R}_{[0,1]}$ is a membership function that describes the degree to which each element of U is a member of the set [115]. Because of the similarity between gray policies and fuzzy sets, much of the work on fuzzy set theory is expected to translate to gray policies. For example, the "very" operator takes a fuzzy set (U, m) and returns the fuzzy set (U, m^2); such an operation is a simple way to make gray policies stricter.

Further generalizations of gray policies may also be possible. For example, rather than the totally ordered set of $\mathbb{R}_{[0,1]}$, gray policies could have complete lattices as their codomains. Some alterations would need to be made to the gray model to handle such codomains, including replacing infimum (supremum) operations with meet (join) operations.

Yet another direction is in the area of enforceability theory. As other work has delineated the black-and-white properties enforceable by various mechanisms (e.g., [101, 64, 70, 38, 12, 37]), the same could be done for gray properties and/or silhouette judges. This direction of research would explore whether, and how well, different mechanisms (static code analyzers or runtime monitors, possibly constrained in various ways) can enforce classes of gray properties and/or silhouette judges.

With regards to injection attacks, it would be interesting to evaluate the accuracy of existing injection-attack-vulnerability databases. With an implementation of the ideas in Chapter 4, it might be straightforward to examine such databases and determine if any of their allegedly malicious inputs are actually benign, and what percentage of their allegedly benign inputs are actually attacks.

Furthermore, the definition of injection attack relies upon taint-tracking, and yet taint-tracking mechanisms are either slow (e.g., bit-level taint tracking), unsound (e.g., limited to tracking taints only within strings), or overly conservative (e.g., static flow analysis). Advances in the realm of taint-tracking would make precisely preventing injection attacks more practical.

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] Mario S. Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey Smith. Additive and multiplicative notions of leakage, and their capacities. In *Proceedings of the Computer Security Foundations Symposium*, pages 308–322, July 2014.

[4] Mario S. Alvim, Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *Proceedings of the Computer Security Foundations Symposium*, pages 265–279, June 2012.

[5] Xiangdong An, Dawn Jutla, and Nick Cercone. Privacy intrusion detection using dynamic bayesian networks. In *Proceedings of the International Conference on Electronic Commerce*, pages 208–215, 2006.

[6] Miguel E. Andrés, Catuscia Palamidessi, Peter van Rossum, and Geoffrey Smith. Computing the leakage of information-hiding systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 373–389. Springer Berlin Heidelberg, 2010.

[7] Yudistira Asnar, Paolo Giorgini, Fabio Massacci, and Nicola Zannone. From trust to dependability through risk analysis. In *Proceedings of the Conference on Availability, Reliability and Security*, pages 19–26, April 2007.

[8] Man Ho Au and Apu Kapadia. PERM: Practical reputation-based blacklisting without TTPs. In *Proceedings of the Conference on Computer and Communications Security*, pages 929–940, 2012.

[9] Man Ho Au, Apu Kapadia, and Willy Susilo. BLACR: TTP-free blacklistable anonymous credentials with reputation. In *Proceedings of the Symposium on Network and Distributed System Security*, 2012.

[10] Davide Balzarotti, Mattia Monga, and Sabrina Sicari. Assessing the risk of using vulnerable components. In *Proceedings of the Workshop on Quality of Protection*, pages 65–77, 2006.

[11] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 12–24, 2007.

[12] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1):3:1–3:26, June 2013.

[13] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[14] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[15] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *Transactions on Information and System Security (TISSEC)*, 13(2):1–39, 2010.

[16] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[17] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative notions of leakage for one-try attacks. *Electronic Notes in Theoretical Computer Science*, 249(0):75–91, 2009. Proceedings of the Conference on Mathematical Foundations of Programming Semantics.

[18] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. *Science of Computer Programming (SCP)*, 75(7):473–495, 2010.

[19] David FC Brewer and Michael J Nash. The chinese wall security policy. In *Proceedings of the Symposium on Security and Privacy*, pages 206–214. IEEE, 1989.

[20] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *SEM '05: Proceedings of the 5th international workshop on software engineering and middleware*, pages 106–113, 2005.

[21] Rainer Bye, Stephan Schmidt, Katja Luther, and Sahin Albayrak. Application-level simulation for network security. In *Proceedings of the International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 33:1–33:10, 2008.

[22] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *Proceedings of the International Conference on Trustworthy Global Computing*, pages 281–300, 2007.

[23] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[24] Han Chen and Pasquale Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, pages 31–40, 2007.

[25] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings of the Symposium on Security and Privacy*, pages 222–230, May 2007.

[26] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. Technical Report RC24190, IBM, 2007.

[27] Richard Chow and Philippe Golle. Faking contextual data for fun, profit, and privacy. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 105–108, 2009.

[28] Kevin Clark, Ethan Singleton, Stephen Tyree, and John Hale. Strata-Gem: Risk assessment through mission modeling. In *Proceedings of the Workshop on Quality of Protection*, pages 51–58, 2008.

[29] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, October 2009.

[30] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010.

[31] Michael R. Clarkson and Fred B. Schneider. Quantification of integrity. *Mathematical Structures in Computer Science*, 25(2):207–258, 2015.

[32] Sebastian Clauß. A framework for quantification of linkability within a privacy-enhancing identity management system. In *Emerging Trends in Information and Communication Security*, volume 3995 of *Lecture Notes in Computer Science*, pages 191–205. 2006.

[33] Sebastian Clauß and Stefan Schiffner. Structuring anonymity metrics. In *Proceedings of the Workshop on Digital Identity Management*, pages 55–62, 2006.

[34] Marc Dacier, Yves Deswarte, and Mohamed Kaniche. Quantitative assessment of operational security: Models and tools, 1996.

[35] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the Symposium on Security and Privacy*, pages 109–124, 2010.

[36] Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, pages 54–68, 2002.

[37] Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, February 2015.

[38] Peter Drábik, Fabio Martinelli, and Charles Morisset. Cost-aware runtime enforcement of security policies. In *Proceedings of the Workshop on Security and Trust Management*, pages 1–16, September 2012.

[39] Peter Drábik, Fabio Martinelli, and Charles Morisset. A quantitative approach for inexact enforcement of security policies. In *Proceedings of the International Conference on Information Security*, pages 306–321, 2012.

[40] M. Edman, F. Sivrikaya, and B. Yener. A combinatorial approach to measuring anonymity. In *Proceedings of the Conference on Intelligence and Security Informatics*, pages 356–363, May 2007.

[41] Barbara Espinoza and Geoffrey Smith. Min-entropy as a resource. *Information and Computation*, 226(0):57–75, 2013. Special Issue: Information Security as a Resource.

[42] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech House, 2003.

[43] P.W.L. Fong. Access control by tracking shallow execution history. In *Proceedings of the Symposium on Security and Privacy*, pages 43–55, 2004.

[44] Marcel Frigault, Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring network security using dynamic bayesian network. In *Proceedings of the Workshop on Quality of Protection*, pages 23–30, 2008.

[45] Jeffrey Gennari and David Garlan. Measuring attack surface in software architecture. Technical Report CMU-ISR-11-121, Institute for Software Research at Carnegie Mellon University, 2012.

[46] Arthur Gervais, Reza Shokri, Adish Singla, Srdjan Capkun, and Vincent Lenders. Quantifying web-search privacy. In *Proceedings of the Conference on Computer and Communications Security*, pages 966–977, 2014.

[47] Joseph A Goguen and José Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*. IEEE Computer Society, 1982.

[48] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the Symposium on Security and Privacy*, pages 575–589, 2014.

[49] I. Goriac. Measuring anonymity with plausibilistic entropy. In *Proceedings of the International Conference on Availability, Reliability and Security*, pages 151–160, Sept 2013.

[50] Vaibhav Gowadia, Csilla Farkas, and Marco Valtorta. PAID: A probabilistic agent-based intrusion detection system. *Computers and Security*, 24(27):529–545, 2005.

[51] Christopher Griffin, Bharat Madan, and Kishor Trivedi. State space approach to security quantification. In *Proceedings of the Conference on Computer Software and Applications Conference*, COMPSAC-W'05, pages 83–88, 2005.

[52] William Halfond, Alex Orso, and Pete Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *Transactions on Software Engineering (TSE)*, 34(1):65–81, 2008.

[53] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, March 2006.

[54] Joseph Y. Halpern and Kevin R. O'Neill. Anonymity and information hiding in multiagent systems. *Journal of Computer Security*, 13(3):483–514, May 2005.

[55] Robert Hansen and Meredith Patterson. Stopping injection attacks with computational theory. In *Black Hat Briefings Conference*, 2005.

[56] Thomas Heumann, Sven Trpe, and Jrg Keller. Quantifying the attack surface of a web application. In *Proceedings of Sicherheit*, volume 170, pages 305–316, 2010.

[57] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137. Springer Berlin Heidelberg, 2005.

[58] Daniel C. Howe and Helen Nissenbaum. TrackMeNot: Resisting surveillance in web search. In *Lessons from the Identity Trail: Anonymity, Privacy, and Identity in a Networked Society*, chapter 23, pages 417–436. 2009.

[59] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006.

[60] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, May 2009.

[61] L. Lamport. *Distributed Systems: Methods and Tools for Specification. An Advanced Course*, volume 190. Springer-Verlag Berlin Heidelberg, May 1985.

[62] A.J. Lee and Ting Yu. Towards quantitative analysis of proofs of authorization: Applications, framework, and techniques. In *Proceedings of the Computer Security Foundations Symposium*, pages 139–153, July 2010.

[63] David John Leversage and Eric James Byres. Estimating a system's mean time-to-compromise. *IEEE Security and Privacy*, 6(1):52–60, January 2008.

[64] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, January 2009.

[65] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proceedings of the European Symposium on Research in Computer Security*, September 2010.

[66] Bev Littlewood, Sarah Brocklehurst, Norman Fenton, Peter Mellor, Stella Page, David Wright, John Dobson, John McDermid, and Dieter Gollmann. Towards operational measures of computer security. *Journal of Computer Security*, 2:211–229, 1993.

[67] Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Automated code injection prevention for web applications. In *Proceedings of the Conference on Theory of Security and Applications*, 2011.

[68] B.B. Madan, K. Gogeva-Popstojanova, K. Vaidyanathan, and K.S. Trivedi. Modeling and quantification of security attributes of software systems. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 505–514, 2002.

[69] Bharat B. Madan, Katerina Goševa-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Perform. Eval.*, 56(1-4):167–186, March 2004.

[70] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, and Jay Ligatti. Enforcing more with less: Formalizing target-aware run-time monitors. In *Proceedings of the International Workshop on Security and Trust Management*, pages 17–32, September 2012.

[71] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, Fabio Martinelli, and Charles Morisset. Probabilistic cost enforcement of security policies. In *Proceedings of the Workshop on Security and Trust Management*, pages 144–159, September 2013.

[72] P.K. Manadhata and J.M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[73] Pratyusa Manadhata, Jeannette Wing, Mark Flynn, and Miles McQueen. Measuring the attack surfaces of two FTP daemons. In *Proceedings of the Workshop on Quality of Protection*, pages 3–10, 2006.

[74] Piotr Mardziel, Mario S. Alvim, Michael Hicks, and Michael R. Clarkson. Quantifying information flow for dynamic secrets. In *Proceedings of the Symposium on Security and Privacy*, pages 540–555, 2014.

[75] Fabio Martinelli, Ilaria Matteucci, and Charles Morisset. From qualitative to quantitative enforcement of security policy. In *Proceedings of the International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, pages 22–35, 2012.

[76] Daryl McCullough. Specifications for multi-level security and a hook-up. In *Proceedings of the Symposium on Security and Privacy*, pages 161–161. IEEE Computer Society, 1987.

[77] Miles A. McQueen, Wayne F. Boyer, Mark A. Flynn, and George A. Beitel. Quantitative cyber risk reduction estimation methodology for a small scada control system. In *Proceedings of the International Conference on System Sciences*, 2006.

[78] Miles A. McQueen, Wayne F. Boyer, Mark A. Flynn, and George A. Beitel. Time-to-compromise model for cyber risk reduction estimation. In *Quality of Protection*, volume 23 of *Advances in Information Security*, pages 49–64. Springer Berlin Heidelberg, 2006.

[79] The MITRE Corporation. *CWE/SANS Top 25 Most Dangerous Software Errors*, 2011. http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

[80] Ian Molloy, Luke Dickens, Charles Morisset, Pau-Chen Cheng, Jorge Lobo, and Alessandra Russo. Risk-based security decisions under uncertainty. In *Proceedings of the Conference on Data and Application Security and Privacy*, pages 157–168, 2012.

[81] Tri Minh Ngo and Marieke Huisman. Quantitative security analysis for multi-threaded programs. In *Proceedings Workshop on Quantitative Aspects of Programming Languages and Systems*, pages 34–48, 2013.

[82] Tri Minh Ngo and Marieke Huisman. Quantitative security analysis for programs with low input and noisy output. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 77–94. Springer Berlin Heidelberg, 2014.

[83] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the International Information Security Conference (SEC)*, pages 372–382, 2005.

[84] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[85] Open Sourced Vulnerability Database. *OSVDB: Open Sourced Vulnerability Database*, 2014. http://osvdb.org/.

[86] Oracle. How to write injection-proof PL/SQL. An Oracle White Paper, 2008. Page 11.

[87] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Trans. Softw. Eng.*, 25(5):633–650, September 1999.

[88] Xinming Ou, S.R. Rajagopalan, and S. Sakthivelmurugan. An empirical approach to modeling uncertainty in intrusion analysis. In *Proceedings of the Computer Security Applications Conference*, pages 494–503, Dec 2009.

[89] The OWASP Foundation. *OWASP Top 10 - 2013*, 2013. http://owasptop10.googlecode.com/files/OWASPTop10-2013.pdf.

[90] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[91] Joseph Pamula, Sushil Jajodia, Paul Ammann, and Vipin Swarup. A weakest-adversary security metric for network configuration security analysis. In *Proceedings of the Workshop on Quality of Protection*, pages 31–38, 2006.

[92] SaiTeja Peddinti and Nitesh Saxena. On the privacy of web search based on query obfuscation: A case study of trackmenot. In *Privacy Enhancing Technologies*, volume 6205 of *Lecture Notes in Computer Science*, pages 19–37. Springer Berlin Heidelberg, 2010.

[93] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 124–145, 2005.

[94] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 179–190, 2012.

[95] Donald Ray and Jay Ligatti. Defining injection attacks. In *Proceedings of the 17th International Information Security Conference (ISC)*, October 2014.

[96] Donald Ray and Jay Ligatti. A theory of gray security policies. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2015.

[97] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *Transactions on Information and System Security*, 1(1):66–92, November 1998.

[98] Pierangela Samarati and SabrinaCapitani de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin Heidelberg, 2001.

[99] Reginald E. Sawilla and Xinming Ou. Identifying critical attack assets in dependency attack graphs. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 18–34, 2008.

[100] Fred B. Schneider. Decomposing Properties into Safety and Liveness using Predicate Logic. Technical Report 87-874, Cornell University, October 1987.

[101] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[102] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. An empirical analysis of input validation mechanisms in web applications and languages. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1419–1426, 2012.

[103] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, pages 41–53, 2003.

[104] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the International Conference on Foundations of Software Science and Computational Structures*, pages 288–302, 2009.

[105] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 1181–1192, 2013.

[106] Ernst Specker. Nicht konstruktiv beweisbare sätze der analysis. *Journal of Symbolic Logic*, 14:145–158, 1949.

[107] Jeffrey Stuckman and James Purtilo. Comparing and applying attack surface metrics. In *Proceedings of the International Workshop on Security Measurements and Metrics*, pages 3–6, 2012.

[108] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 372–382, 2006.

[109] The Open Web Application Security Project (OWASP). SQL injection prevention cheat sheet, 2012. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[110] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[111] Li Xi and Dengguo Feng. FARB: Fast anonymous reputation-based blacklisting without TTPs. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 139–148, 2014.

[112] Li Xi, Jianxiong Shao, Kang Yang, and Dengguo Feng. ARBRA: Anonymous reputation-based revocation with efficient authentication. In *Proceedings of the Conference on Information Security*, volume 8783 of *Lecture Notes in Computer Science*, pages 33–53. Springer Berlin Heidelberg, 2014.

[113] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*, pages 121–136, 2006.

[114] Kin Ying Yu, Tsz Hon Yuen, Sherman S.M. Chow, Siu Ming Yiu, and Lucas C.K. Hui. PE(AR)^2: Privacy-enhanced anonymous authentication with reputation and revocation. In *Proceedings of the European Symposium on Research in Computer Security*, volume 7459 of *Lecture Notes in Computer Science*, pages 679–696. Springer Berlin Heidelberg, 2012.

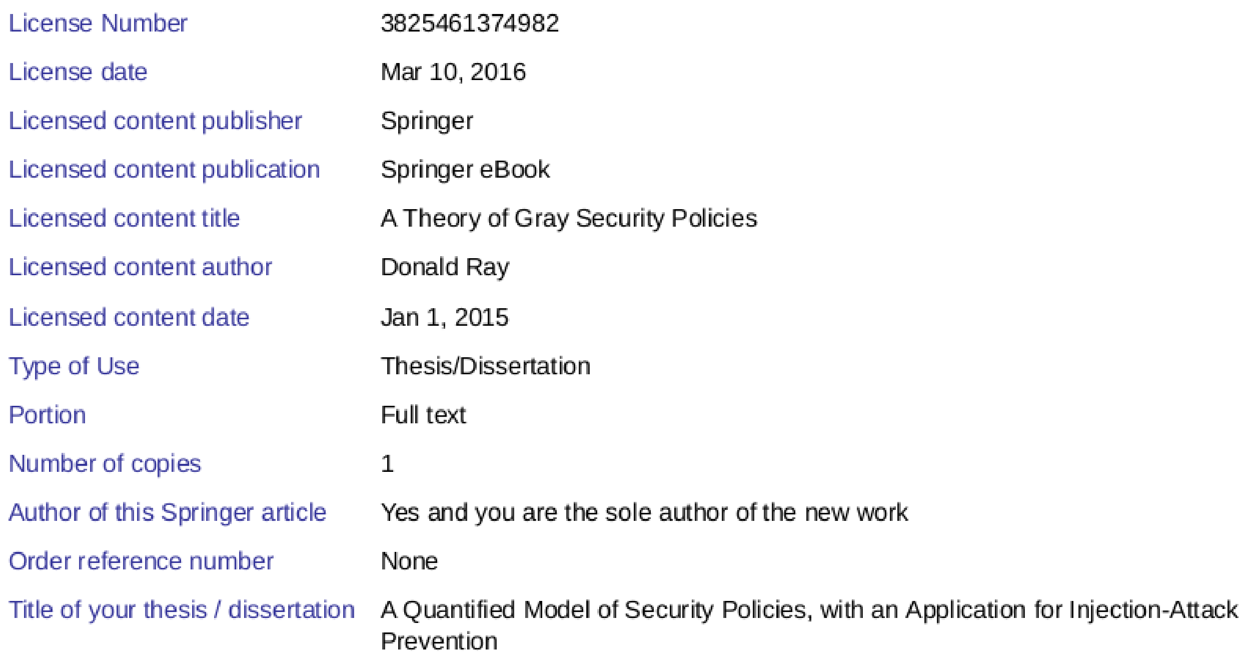
[115] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[116] Rui Zhuang, Su Zhang, A. Bardas, S.A. DeLoach, Xinming Ou, and A. Singhal. Investigating the application of moving target defenses to network security. In *Proceedings of the International Symposium on Resilient Control Systems*, pages 162–169, Aug 2013.

APPENDICES

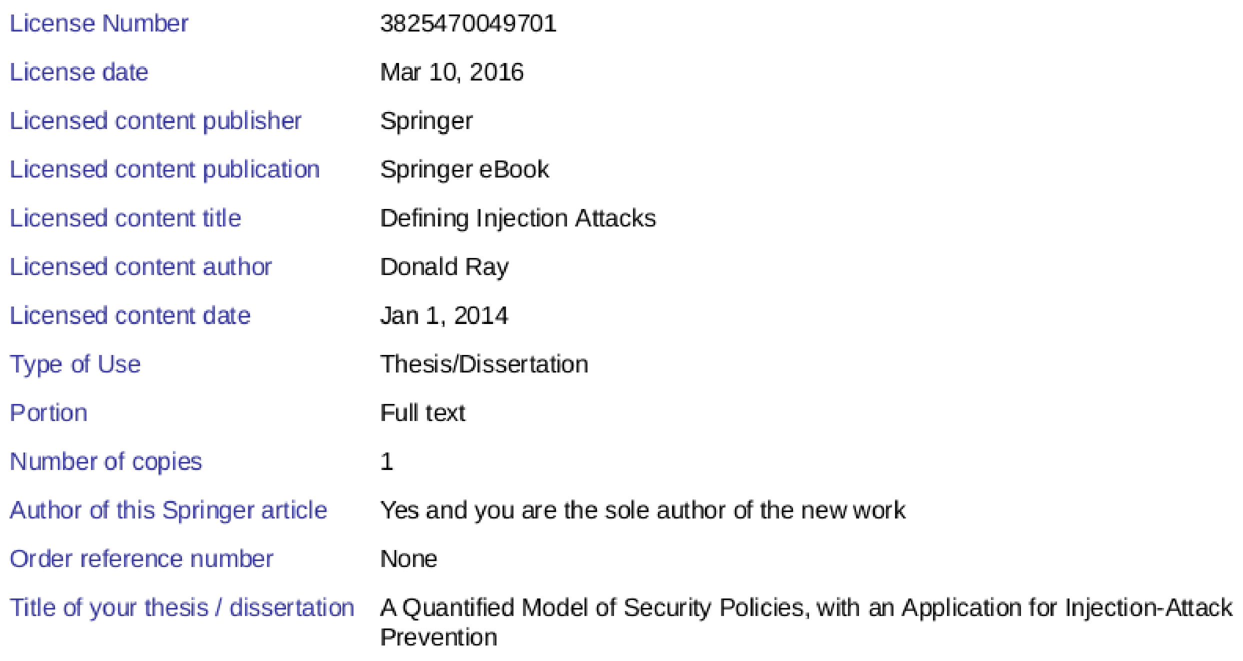
Appendix A Copyright Permissions

The following is permission to use the content in Chapters 2 and 3.



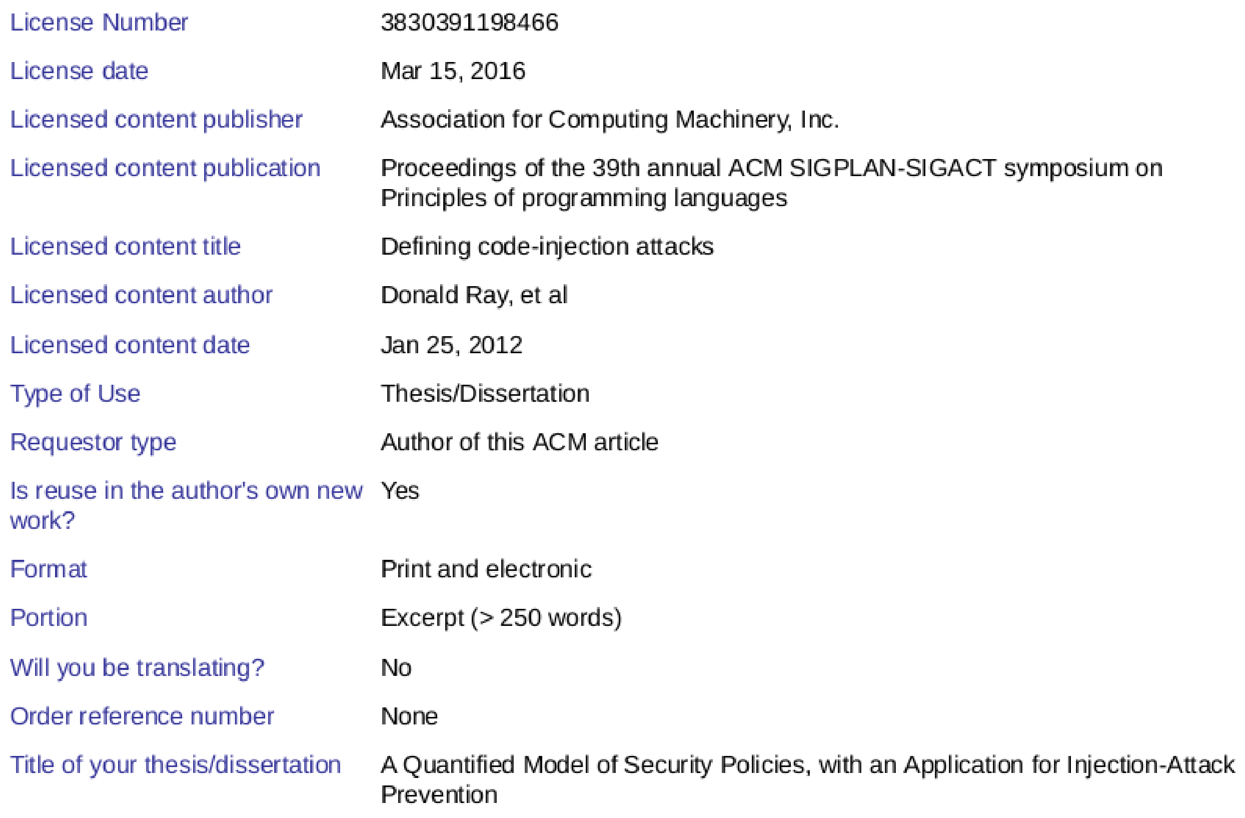
License Number	3825461374982
License date	Mar 10, 2016
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	A Theory of Gray Security Policies
Licensed content author	Donald Ray
Licensed content date	Jan 1, 2015
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	None
Title of your thesis / dissertation	A Quantified Model of Security Policies, with an Application for Injection-Attack Prevention

The following two permissions are for the content in Chapter 4.



License Number	3825470049701
License date	Mar 10, 2016
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Defining Injection Attacks
Licensed content author	Donald Ray
Licensed content date	Jan 1, 2014
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	None
Title of your thesis / dissertation	A Quantified Model of Security Policies, with an Application for Injection-Attack Prevention

Appendix A (Continued)



License Number	3830391198466
License date	Mar 15, 2016
Licensed content publisher	Association for Computing Machinery, Inc.
Licensed content publication	Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
Licensed content title	Defining code-injection attacks
Licensed content author	Donald Ray, et al
Licensed content date	Jan 25, 2012
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Excerpt (> 250 words)
Will you be translating?	No
Order reference number	None
Title of your thesis/dissertation	A Quantified Model of Security Policies, with an Application for Injection-Attack Prevention