A Fragility Metric for Software Diversity

Nilofar Mansourzadeh*, Anil Somayaji*, Jason Jaskolka†
*School of Computer Science
†Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive, Ottawa ON K1S 5B6, Canada
Emails: NilofarMansourzadeh@cmail.carleton.ca, soma@ccsl.carleton.ca, jason.jaskolka@carleton.ca

***Abstract*—Large-scale attacks compromise populations of systems by exploiting vulnerabilities in shared components. This paper presents a novel security metric, population fragility, that captures the potential impact of vulnerable components on a population of systems. We define population fragility using a product family algebra (PFA)-based model that captures how components are in turn made of other components. While our current model has a number of simplifying assumptions, population fragility potentially motivates new security approaches and provides deeper insight into the relationship between diversity and computer security.**

***Index Terms*—fragility, security metrics, software diversity, product family algebra**

I. Introduction

Software vulnerabilities are the dominant concern in most efforts to improve software security today, motivating multiple approaches to the challenge. Software development practices such as the the Microsoft SDL [1] reduce the number and scope of software vulnerabilities through careful design, security audits, vulnerability scanning, and other techniques. Ubiquitous software update mechanisms allow for vulnerable applications to be patched after they have been deployed. Firewalls, anti-malware systems, and even mandatory access control mechanisms (as configured on current systems) all attempt to reduce the impact of vulnerabilities that remain. Despite all of these efforts, vulnerabilities continue to be found and exploited by attackers.

Rather than focusing on eliminating vulnerabilities, another strategy is to address the factors that make vulnerabilities so dangerous. The impact of any vulnerability arises from both its severity and its frequency. A vulnerability that allows for total system compromise is not much of a concern if it only affects a few systems (assuming those systems are not particularly high value targets). A vulnerability that exists on millions of systems, however, is a real threat.

Diversity has been proposed as a strategy for reducing the impact of vulnerabilities [2]. The idea is that if systems are different, then vulnerabilities will only impact a subset of systems at any time. Many researchers have criticized software monocultures—environments with minimal software diversity—as being especially insecure, especially in the context of dominant operating systems [3]. Software monocultures, however, have significant benefits in terms of development time and administrative overhead, as fewer types of systems translates to fewer systems for administrators to learn how to configure, maintain, and secure.

Given that software diversity has both clear benefits and costs, a natural approach to improving security would be to devise measures of both in order to determine an appropriate trade-off given available resources and expected risks. While there are numerous security metrics [4], [5], [6], [7], [8], the choices for measuring software diversity are much rarer, and most of these do not measure diversity in a way that is relevant to security.

Here we propose a new metric for software diversity, population fragility, that captures a key factor relevant to security: the impact of a vulnerability on a population of systems. The insight behind our fragility metric is that as software systems are created through the composition of components, this nesting structure of components should be taken into account when assessing software diversity for security. The severity of a vulnerability is proportional to the number of vulnerable components in a population, whether that component is a function, library, application, container, virtual machine, or host. So, if we can model the relative commonality of components at multiple scales, we can devise a measure of the potential impact of an arbitrary vulnerability.

This paper presents a simple analytical feature model based on product family algebra for defining and studying the distribution of components within computer systems in a population. This model is highly idealized; in particular, it assumes a uniform distribution of component configurations. The model is sufficient, however, to allow us to define a security-oriented diversity metric that captures the impact of vulnerabilities in shared components.

We can summarize our work as follows. We model a population as a set of systems that each are composed of a set of components, and each of these components are made up of further sub-components. Each component can have one or more variations. We can thus model hardware-level features such as CPUs, software components such as operating systems, applications, and libraries, and algorithmic components such as cryptographic primitives and protocols. Vulnerabilities are defined in the context of affected components. If a system has a vulnerable component, it is said to be vulnerable. The fragility of a population, relative to a vulnerability, is then the fraction of the population that could be compromised by that vulnerability. A population that can be completely compromised by one vulnerability is maximally fragile; to minimize fragility, we minimize the fraction of a population that can be compromised by a vulnerability. The rest of this

presents a formal version of this model along with worked examples.

The remainder of this paper is organized as follows. Section II provides the mathematical background describing product family algebra, products and features. Section III describes our approach for modeling populations of computer systems using product family algebra. Section IV uses the model to define and analyze population fragility as a security metric. We discuss related work in Section V. Section VI concludes with a discussion of contributions, limitations, and future work.

II. Mathematical Background

In this section, we introduce the mathematical background of product family algebra and a number of related concepts including products and features which are needed for the development of our analytical framework for modeling populations of computer systems in Sections III and IV.

A. Product Family Algebra

Product family algebra (PFA) [9] is an algebraic feature modeling technique with the power to describe product families precisely. It is based on the mathematical structure of idempotent commutative semirings. In addition, it allows algebraic calculations and manipulations of product families to generate new information about those product families.

A *semiring* is a mathematical structure $\big(S, +, \cdot, 0, 1\big)$ where $\big(S, +, 0\big)$ is a commutative monoid and $\big(S, \cdot, 1\big)$ is a monoid such that operator $\cdot$ distributes over operator $+$ and element 0 is annihilates S with respect to $\cdot$. We say that a semiring is *idempotent* if operator $+$ is idempotent (i.e., $x + x = x$). We say that a semiring is *commutative* if operator $\cdot$ is commutative (i.e., $x \cdot y = y \cdot x$).

Definition 1 (Product Family Algebra)**.** *A product family algebra (PFA) is an idempotent and commutative semiring* $\big(S, +, \cdot, 0, 1\big)$ *where each element of the semiring is a product family.*

In the product family context, $+$ can be interpreted as a choice or option between two product families and $\cdot$ can be interpreted as a mandatory composition of two product families. The constant 0 represents the empty family and the constant 1 represents the family that has one product without features. The term $a + 1$ is the product family offering the choice between a and the identity product and indicates that the feature a is optional.

With the above interpretations, other concepts in product family modeling can be expressed mathematically. In general, each idempotent semiring $\big(S, +, \cdot, 0, 1\big)$ has a natural partial order $\leq$ on S defined by $a \leq b \iff a + b = b$. Therefore, for product families $a, b \in S$, $a \leq b$ indicates that a is a *sub-family* of b if and only if $a + b = b$.

B. Products and Features

The basic building blocks of a product family in PFA are *products* and *features*.

Definition 2 (Product)**.** *We say that* a *is a product if it is different than* 0 *and satisfies:*

$$\forall(b \mid: b \leq a \implies (b = 0 \ \vee \ b = a)) \tag{1}$$

$$\forall(b, c \mid: a \leq b + c \implies (a \leq b \ \vee \ a \leq c)) \tag{2}$$

Equation 1 shows that a product does not have a subfamily except the empty family and itself. Equation 2 indicates that if a product a is a subfamily of a family formed by c and b, it must be a subfamily of one of them. Intuitively, this indicates that a product cannot be split using the choice operator $+$.

Definition 3 (Feature)**.** *We say that* a *is a feature if it is a proper product different than* 1 *satisfying:*

$$\forall(b \mid: b \leq a \implies (b = 1 \ \vee \ b = a)) \tag{3}$$

$$\forall(b, c \mid: a | b \cdot c \implies (a | b \ \vee \ a | c)) \tag{4}$$

where the division operator $|$ *is defined by* $a | b \iff \exists(c \mid: b = a \cdot c)$.

Equation 3 states that if we have a product b that divides a, then either b is 1 or $b = a$. Equation 4 states that for all product families b and c, if a is mandatory to form $b \cdot c$, then it is mandatory to form b or it is mandatory to form c. Intuitively, this indicates that a feature cannot be split using the composition operator $\cdot$.

New product families can be derived from other existing product families by adding features. The *refinement* relation captures such a relationship between two product families.

Definition 4 (Refinement)**.** *The refinement relation on a PFA is defined as follows:*

$$a \sqsubseteq b \iff \exists(c \mid c \in S \ : \ a \leq b \cdot c)$$

Informally, a product family a refines another product family b if a has the same set of features as b and possibly more. For example, assume that we have a $\mathrm{new\_mobile}$ that has screen, keypad, $\mathrm{calling\_feature}$, and GPS. We have also an $\mathrm{old\_mobile}$ that has screen, keypad, and $\mathrm{calling\_feature}$. Therefore, $\mathrm{new\_mobile}$ refines $\mathrm{old\_mobile}$ because every product in $\mathrm{new\_mobile}$ has all features of some products in $\mathrm{old\_mobile}$. When a product a refines a product b, we say that b is a *sub-product* of a.

III. Modeling

In this section, we develop a model of computer systems and their variants for different distributions within a population of users. To achieve this, we use product family algebra (PFA). PFA helps to capture and analyze the commonalities and variabilities of a product family and allows mathematical description and manipulation of product family specifications. We use this model in Section IV to define population fragility.

A. An Example Population of Computer Systems

For simplicity, suppose that a computer system is comprised of hardware (hw) and software (sw). The hardware for the system involves only a central processing unit (CPU), motherboard (mb), random access memory (RAM), and a

hard drive (hd). The software for the system involves only an operating system (OS) and an optional application (app). For any computer system, there are two kinds of CPU, mb, RAM, and hd, and three kinds of OS and three kinds of app. The product family of computer systems can be visualized as a graphical feature model as shown in Figure 1.

To study populations of computer systems, we assume that we have a set of computer system users. Each user operates a computer system that can be built from the product family described above. In this paper, we assume that the products that can be built from this product family are distributed uniformly among the population of users. This assumption enables us to focus on the impact of sharing vulnerable components in a population.

B. Specifying the Computer System Product Family using PFA

We use PFA to specify the product family described in Section III-A and illustrated in Figure 1. We begin by declaring the basic features of computer systems. The basic features represent all of the possibly components that can be used to build a computer system. In our example, we have 14 basic features corresponding to the specific kinds of CPU, mb, RAM, hd, OS and app. Then, using the basic features and the operators of PFA, we define a labeled product family specifying the mandatory and optional features of products. For example, as described in Section III-A, the software for a computer system requires only an operating system and an optional application. This is represented as $\mathrm{sw} = \mathrm{OS} \cdot (\mathrm{app} + 1)$ indicated that it is mandatory to have an operating system and optional to have application. Subsequently, because we have three kinds of operating system that we can choose from, we specify $\mathrm{OS} = \mathrm{Windows} + \mathrm{MacOS} + \mathrm{Linux}$ to show that an operating system is one of Windows, MacOS or Linux. The complete PFA specification for our example computer system product family is shown in Figure 2. We will use this PFA specification of the computer system product family to evaluate the impact of a vulnerable component on the security of an entire population of computer systems from this product family.

C. Computing the Size of the Catalog of Products

Given the PFA specification of a product family, such as that shown in Figure 2, we can compute the total number of unique products that can be built from the specification.

Definition 5 (Catalog Size)**.** *Let C be a product family. Then, the number of products that can be built from C is called the* catalog size *(denoted $|C|$) and is computed recursively on the structure of product family algebra:*

$$\begin{aligned} |0| &= 0 \\ |1| &= 1 \\ |a| &= 1 \quad \textit{(a is a basic feature)} \\ |a+b| &= |a| + |b| \\ |a \cdot b| &= |a| \times |b| \end{aligned}$$

For the sake of consistency we will refer to the set of unique products that can be built from the specification of a product family as the *catalog*.

Example 1 (Computing the Catalog Size)**.** *Applying Definition 5 to the PFA specification of our computer system product family in Figure 2, we have* 192 *possible computer system products in the catalog.*

$$\begin{aligned} |\mathrm{Computer}| &= |\mathrm{hw} \cdot \mathrm{sw}| \\ &= |\mathrm{hw}| \times |\mathrm{sw}| \\ &= |\mathrm{CPU} \cdot \mathrm{mb} \cdot \mathrm{RAM} \cdot \mathrm{hd}| \times |\mathrm{OS} \cdot (\mathrm{app}+1)| \\ &= [(1+1) \times (1+1) \times (1+1) \times (1+1)] \times \\ &\quad [(1+1+1) \times (1+1+1)+1] \\ &= [2 \times 2 \times 2 \times 2] \times [3 \times 4] \\ &= 16 \times 12 \\ &= 192 \end{aligned}$$

■

It is important to understand how the addition of new alternative products or features (i.e., variations) in a product family affects the catalog size. The following proposition shows that adding non-zero variations to a product family increases the *catalog size*.

Proposition 1 (Variations Increase the Catalog Size)**.** *Adding non-zero variations to a product family increases the size of the catalog.*

Proof. Let C be the original product family and let C' be the product family after adding a non-zero variation. A variation is the addition of an alternative feature that allows two products to differ in the choice of that feature. Then,

$$\begin{aligned} & |C| < |C'| \\ \Longleftarrow\ & \langle\ \text{Hypothesis: } C' \text{ is the product family } C \text{ with an additional product or feature } v \neq 0\ \rangle \\ & |C| < |C + v| \\ \Longleftrightarrow\ & \langle\ \text{Definition 5}\ \rangle \\ & |C| < |C| + |v| \\ \Longleftrightarrow\ & \langle\ \text{Arithmetic} \quad \& \quad |v| > 0\ \rangle \\ & \mathsf{true} \end{aligned}$$

□

Intuitively, Proposition 1 states that if we provide more choices to build a computer system, we can build more unique products that can be distributed among our population of users.

IV. Fragility Analysis

In the previous section, we developed a model of a product family of computer systems that are distributed among a population of users. Using this model, we now study the *population fragility* using these computer systems. In this section, we define a measure of the *population fragility* with

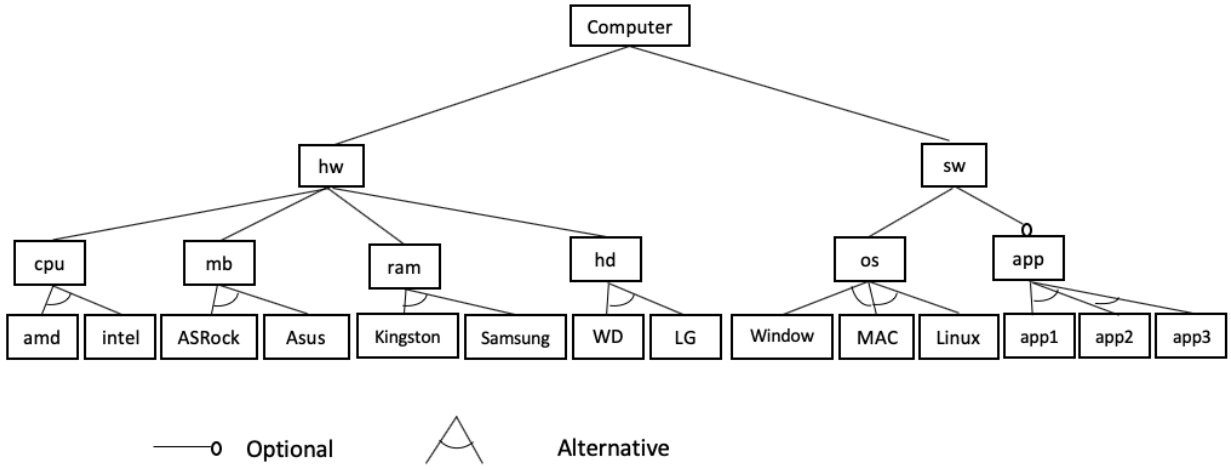

Fig. 1. Feature model for a computer system

```
% Declarations of basic features
bf amd
bf intel
bf ASRock
bf Asus
bf Kingston
bf Samsung
bf WD
bf Seagate
bf Windows
bf MacOS
bf Linux
bf app1
bf app2
bf app3
```

```
% Definitions of labeled product family
Computer = hw · sw

      hw = CPU · mb · RAM · hd
      sw = OS · (app + 1)

     CPU = amd + intel
      mb = ASRock + Asus
     RAM = Kingston + Samsung
      hd = WD + Seagate

      OS = Windows + MacOS + Linux
     app = app1 + app2 + app3
```

Fig. 2. A PFA specification of the computer system product family

respect to a known exploitable system component. Then, we study how increasing variability in the product family can improve the *population fragility*.

A. Defining Population Fragility

Because products within a product family contain commonalities, there is a potential that multiple products contain the same exploitable component or sub-product. To study this phenomenon, we aim to define a measure to show how much of a population is susceptible to an attack when we know that there exists an exploitable component in the computer system product family. To do so, we need to identify which of the products in a product family contain an exploitable sub-product; that is, the set of products which contain a vulnerability that can be exploited by an adversary to conduct an attack on the system. We call this set of products the *set of exploitable products*.

To determine set of exploitable products, we assume that there is an exploitable sub-product that we know about beforehand. Using this information, we find the set of products in the catalog that contain these exploitable sub-products. By determining the number of exploitable products with respect to the total number of products that can be built from the product family (i.e., the catalog size as defined in Section III-C), we can determine the proportion the population[1] that is susceptible to an attack on the known exploitable sub-product. We call this measure the *population fragility*. Formally, the measure of the *population fragility* is defined as follows:

Definition 6 (Population Fragility)**.** *Let C be a product family and let x be an exploitable sub-product. Then, the fragility of C with respect to x is given by:*

$$\text{Fragility}(C, x) = \frac{|X|}{|C|}$$

where $X = \{c \mid c \in C \ \wedge \ c \sqsubseteq x\}$ is the set of exploitable products in the product family C.

For the sake of our example, suppose that it has been revealed that there is an exploitable vulnerability affecting

[1]Note that because we assume that the products in the catalog are uniformly distributed among all users within a population, we can view the catalog size and the size of the population as being equal.

computer systems containing the combination of Windows, intel, and app1. Because we do not know exactly which of the features Windows, intel, and app1 has the exploitable vulnerability—it may be one of them or any combination of them—we say that all computer systems that contain the exploitable sub-product (Windows · intel · app1) are vulnerable. Applying Definition 6 shows that, out of 192 possible computer system products, there are 8 products that contain the known exploitable sub-product. This means that the *population fragility* is 0.0417 as detailed in Example 2 below.

Example 2 (Population fragility with respect to the exploitable sub-product Windows · intel · app1)**.** *Applying Definition 6, the set of exploitable products is given by:*

$$\begin{aligned} X \;=\; \{&(\text{intel} \cdot \text{ASRock} \cdot \text{Kingston} \cdot \text{WD} \cdot \text{Windows} \cdot \text{app1}),\\ &(\text{intel} \cdot \text{Asus} \cdot \text{Samsung} \cdot \text{Seagate} \cdot \text{Windows} \cdot \text{app1}),\\ &\ldots,\\ &(\text{intel} \cdot \text{Asus} \cdot \text{Samsung} \cdot \text{WD} \cdot \text{Windows} \cdot \text{app1})\} \end{aligned}$$

Therefore, $|X| = 8$*. Using the results from Example 1, we can compute the fragility of* C *with respect to* x*:*

$$\text{Fragility}(C, x) = \frac{|X|}{|C|} = \frac{8}{192} = 0.0417$$

■

Now that we are able to compute the *population fragility*, we turn our attention to determining how we can improve the *population fragility* by adding variations to the product family of computer systems distributed among the population.

B. Adding Variations to Exploitable Products

As shown in Proposition 1, adding variations to a product family specification can increase the size of the catalog of products that can be built. This means that we can have more products with more variability, meaning that it is less likely that products share combinations of features. When considering how we can improve the *population fragility*, there are several places where we can add variations. One of these places is in the labeled product families that contain the exploitable sub-product. To illustrate our intuition, we carry out a broad example of adding variations in this manner. The details are described in Example 3 below.

Example 3 (Adding variation to labeled product families that contain the exploitable sub-product)**.** *We continue to assume that we have the exploitable sub-product* $x = (\text{Windows} \cdot \text{intel} \cdot \text{app1})$*.Consider the addition of one more alternative to each of* OS*,* CPU*, and* app *labeled product families in the PFA specification shown in Figure 2. More specifically, suppose we add* Unix *as an alternative operating system,* arm *as an alternative* CPU*, and* app4 *as an alternative* app*. Note that these additions yield new choices to avoid the features present in the exploitable sub-product. The revised PFA specification for our example computer system product family with the added variations is shown in Figure 3.*

As a result, the set of exploitable products (i.e., X*) is the same as in Example 2. Therefore,* $|X|$ *remains* 8*, while the catalog size (i.e.* $|C|$*), computed using Definition 5, increases from* 192 *to* 480*. Thus, by applying Definition 6, the population fragility is reduced to* 0.0167 *as a result of these added variations in the product family.* ■

The following proposition generalizes our intuition that adding variations to exploitable products reduces the *population fragility*.

Proposition 2 (Adding Variation to Exploitable Products)**.** *Adding non-zero variations in an exploitable product decreases the population fragility.*

Proof. Assume an exploitable sub-product x. Let $\text{Fragility}(C, x)$ be the original *population fragility* and let $\text{Fragility}(C', x)$ be the *population fragility* after adding a non-zero variation in an exploitable product.

$$\begin{aligned} &\text{Fragility}(C, x) > \text{Fragility}(C', x)\\ \iff &\langle\ \text{Definition 6}\ \rangle\\ &\frac{|X|}{|C|} > \frac{|X'|}{|C'|}\\ \impliedby &\langle\ \text{Hypothesis: } X \subseteq X' \Longrightarrow |X'| = |X' \backslash X| + |X|\ \rangle\\ &\frac{|X|}{|C|} > \frac{|X' \backslash X| + |X|}{|C'|}\\ \iff &\langle\ \text{Fraction Addition}\ \rangle\\ &\frac{|X|}{|C|} > \frac{|X' \backslash X|}{|C'|} + \frac{|X|}{|C'|}\\ \impliedby &\langle\ \text{Hypothesis: Add variation in an exploitable sub-product} \implies \neg\, \exists (c \mid c \in C' \backslash C : c \sqsubseteq x) \implies |\{c \mid c \in C \wedge c \sqsubseteq x\}| = |\{c \mid c \in C' \wedge c \sqsubseteq x\}| \implies |X| = |X'| \implies |X' \backslash X| = 0\ \rangle\\ &\frac{|X|}{|C|} > \frac{|X|}{|C'|}\\ \iff &\langle\ \text{Proposition 1: } |C| < |C'|\ \rangle\\ &\text{true} \end{aligned}$$

□

Proposition 2 shows that if we can decrease the likelihood of an exploitable sub-product being shared among a large proportion of the population, then we can improve the *population fragility*. The results of Proposition 2 show that this can be achieved by providing more alternatives to avoid known combinations of vulnerable system components that so that a smaller proportion of the population shares these vulnerabilities.

C. Adding Variations to Non-exploitable Products

In the previous section, we added variations in the labeled product families that contain the exploitable sub-product. Here we explore the effect that adding variations in the non-exploitable sub-products has on the *population fragility*. As in the previous section, we being with a broad example of adding variations in this manner to illustrate our intuition. The details are described in Example 4 below.

```
% Declarations of basic features
bf amd
bf intel
bf arm
bf ASRock
bf Asus
bf Kingston
bf Samsung
bf WD
bf Seagate
bf Windows
bf MacOS
bf Linux
bf Unix
bf app1
bf app2
bf app3
bf app4
```

```
% Definitions of labeled product family
Computer = hw · sw

      hw = CPU · mb · RAM · hd
      sw = OS · (app + 1)

     CPU = amd + intel + v(arm)
      mb = ASRock + Asus
     RAM = Kingston + Samsung
      hd = WD + Seagate

      OS = Windows + MacOS + Linux + Unix
     app = app1 + app2 + app3 + app4
```

Fig. 3. Revised PFA specification with new variations (emphasize in boldface) in labeled product families that contain the exploitable sub-product $x = (\text{Windows} \cdot \text{intel} \cdot \text{app1})$

Example 4 (Adding variation to labeled product families that do not contain the exploitable sub-product)**.** *Again we assume that we have the exploitable sub-product* $x = (\text{Windows} \cdot \text{intel} \cdot \text{app1})$. *Consider the addition of one more alternative to each of the labelled product families* CPU, mb, RAM, hd, OS, *and* app *in the PFA specification shown in Figure 2. The revised PFA specification is similar to that shown in Figure 3. Note that that only do these additions overlap with the exploitable sub-product* $x = (\text{Windows} \cdot \text{intel} \cdot \text{app1})$, *but also with the non-exploitable sub-products in the product family.*

As a result of the additions, the number of exploitable products (i.e., $|X|$*) increases from* 8 *to* 27 *and the catalog size (i.e.,* $|C|$*) increases from* 192 *to* 1620*. Applying Definition 6, we find that the population fragility is* 0.0167 *again.* ■

In Example 4, the *population fragility* remains unchanged because the ratio between the number of exploitable products and the catalog size remains same as that in Example 3. Adding more variations to the non-exploitable products (e.g., mb, RAM, and hd), increases the number of products that contain the exploitable sub-product while also increasing the catalog size; we can build new products, but a subset of those new products inevitably contain the exploitable sub-product and thus become exploitable products themselves. Therefore, increasing variations in the non-exploitable sub-products does not help to reduce the *population fragility*. The following proposition generalizes these observations:

Proposition 3 (Adding Variation to Non-exploitable Products)**.** *Adding non-zero variations in a non-exploitable sub-product does not change the population fragility.*

Proof. Assume an exploitable sub-product x. Let $\text{Fragility}(C, x)$ be the original *population fragility* and let $\text{Fragility}(C', x)$ be the *population fragility* after adding a non-zero variation in a non-exploitable sub-product.

$$\text{Fragility}(C, x) = \text{Fragility}(C', x)$$

$$\iff \langle \text{ Definition 6 } \rangle$$
$$\frac{|X|}{|C|} = \frac{|X'|}{|C'|}$$
$$\impliedby \langle \text{ Multiply Both Sides by 2 } \rangle$$
$$2\frac{|X|}{|C|} = 2\frac{|X'|}{|C'|}$$
$$\iff \langle \text{ Multiply Both Sides by } 1 = \tfrac{|C'|}{|C'|} = \tfrac{|C|}{|C|} \rangle$$
$$2\frac{|X||C'|}{|C||C'|} = 2\frac{|X'||C|}{|C'||C|}$$
$$\iff \langle \text{ Expand Multiplication: } 2a = a + a \rangle$$
$$\frac{|X||C'| + |X||C'|}{|C||C'|} = \frac{|X'||C| + |X'||C|}{|C'||C|}$$
$$\iff \langle \text{ Fraction Addition } \rangle$$
$$\frac{|X||C'|}{|C||C'|} + \frac{|X||C'|}{|C||C'|} = \frac{|X'||C|}{|C'||C|} + \frac{|X'||C|}{|C'||C|}$$
$$\iff \langle \text{ Arithmetic } \rangle$$
$$\frac{|X||C'|}{|C||C'|} - \frac{|X'||C|}{|C'||C|} = \frac{|X'||C|}{|C'||C|} - \frac{|X||C'|}{|C||C'|}$$
$$\iff \langle \text{ Subtract Both Sides by } \tfrac{|C'||C|}{|C'||C|} \rangle$$
$$\frac{|X||C'|}{|C||C'|} - \frac{|C'||C|}{|C'||C|} - \frac{|X'||C|}{|C'||C|} = \frac{|X'||C|}{|C'||C|} - \frac{|C||C'|}{|C||C'|} - \frac{|X||C'|}{|C||C'|}$$
$$\iff \langle \text{ Arithmetic \& Distributivity } \rangle$$
$$\frac{|X||C'|}{|C||C'|} - \frac{\big(|C'| - |X'|\big)|C|}{|C'||C|} = \frac{|X'||C|}{|C'||C|} - \frac{\big(|C| - |X|\big)|C'|}{|C||C'|}$$
$$\iff \langle \text{ Cancellation } \rangle$$
$$\frac{|X|}{|C|} - \frac{|C'| - |X'|}{|C'|} = \frac{|X'|}{|C'|} - \frac{|C| - |X|}{|C|}$$
$$\iff \langle \text{ Arithmetic } \rangle$$
$$\frac{|X|}{|C|} + \frac{|C| - |X|}{|C|} = \frac{|X'|}{|C'|} + \frac{|C'| - |X'|}{|C'|}$$
$$\impliedby \langle \text{ Hypothesis: } X \subseteq C \implies |C \backslash X| = |C| - |X| \ \wedge \ X' \subseteq C' \implies |C' \backslash X'| = |C'| - |X'| \rangle$$

$$\frac{|X|}{|C|} + \frac{|C\backslash X|}{|C|} = \frac{|X'|}{|C'|} + \frac{|C'\backslash X'|}{|C'|}$$
$$\iff \langle \text{ Fraction Addition } \rangle$$
$$\frac{|X| + |C\backslash X|}{|C|} = \frac{|X'| + |C'\backslash X'|}{|C'|}$$
$$\Longleftarrow \langle \text{ Hypothesis: } X \subseteq C \implies |C| = |C\backslash X| + |X| \ \wedge \ X' \subseteq C' \implies |C'| = |C'\backslash X'| + |X'| \ \rangle$$
$$\frac{|C|}{|C|} = \frac{|C'|}{|C'|}$$
$$\iff \langle \text{ Cancellation } \quad \& \quad |C| \neq 0 \quad \& \quad |C'| \neq 0 \ \rangle$$
$$1 = 1$$
$$\iff \langle \text{ Reflexivity of } = \rangle$$
$$\text{true}$$

□

Proposition 3 emphasizes the point that not all variations are effective at reducing the *population fragility*. This is important because a tendency may be to simply add as many variations as feasible into the product family. We need to be careful to not simply build more products that contain exploitable subproducts. The decision of where to introduce these variations needs to be much more strategic as indicated by the results of Proposition 2.

V. Related Work

Security metrics is a complex topic, with books [4], [5] and surveys of different areas including systems security metrics [6], network security metrics [7], embedded security metrics [8] among others. Despite this variety, the core objective of employing security metrics within an organization is to establish a tangible and measurable way to assess the cybersecurity posture. This quantification allows organizations to gauge how secure their systems are, with a higher metric indicative of a robust defense mechanism making cyberattacks arduous. The large number of metrics arises from the numerous ways security can be quantified, capturing virtually any aspect of system configuration or history that could be related to security. Some measures are historical, such as how often vulnerabilities have been found, how long it took for vulnerabilities to be patched, and the time between patches being made available to being applied to a given system. Some are empirical, arising from lab or field studies where a defense system's ability to detect malware or intrusions are tested. Others are more theoretical, such as password strength or address space randomization entropy, where the construction of the system should give certain security properties, but its real-world security is impacted by many external factors. (Cryptographic security measures are virtually all theoretical.) No matter the type of metrics, however, each is only capturing a small aspect of system security.

Most past work in metrics for software diversity is focused on diversity from the perspective of fault tolerance [10]. Larsen (2014) noted that the field of automated software diversity needed better metrics [11]. Subsequent to this we know of two works that present diversity metrics for security, Zhang (2016) [12] and Tong (2019) [13]. Both works are inspired by diversity metrics used in ecology, unlike ours which is inspired by work in software engineering. The focus of Zhang's work is on network-level diversity using an attack graph-related approach; however, their work, although they also examine how to determine resource similarity using file and modification-level similarity [12]. In contrast, Tong's work presents an attribute matrix-based metric that captures variations such as differences in hardware, operating system, and applications [13]. The key distinguishing characteristics of our work are 1) the focus on components and component aggregations that can be modeled using PFA and not previous approaches and 2) a precise connection between our model, metric, and security, namely impact of vulnerabilities in shared components.

Feature Models (FMs) were first introduced and used by Kang et al. in 1990 [14]. They have since become well accepted and applied in both academic and industrial projects. A feature model represents a combination of system characteristics called features such that each combination corresponds to a member in a product family. A product family is a group of products that normally share common features. Moreover, it describes the commonalities, variabilities, and dependencies between features. The main challenges of feature modeling include the development, maintenance, and evolution of complex and large feature models. Specifically, it is difficult to handle unanticipated changes in large feature models [15].

A number of notable feature modeling methodologies and notations have been proposed: Feature Oriented Domain Analysis (FODA) [14], [16], Feature-Oriented Reuse Method (FORM) [17], Reuse-Driven Software Engineering Business (RSEB) [18], Featured Reuse-Driven Software Engineering Business (FeatuRSEB) [19], among others. The key advantage of Product Family Algebra (PFA) [9] over other product family specification formalisms, such as FODA and FORM, is that PFA can draw upon the large body of theoretical results for idempotent commutative semiring and for algebraic techniques in general, which help improve the consistency, correctness, compatibility, and reusability of PFA-based models.

VI. Discussion and Concluding Remarks

The key contribution of our population fragility metric is that it highlights the potential risks of shared components. In past works criticizing software monocultures there is a focus on monopolies, particularly in operating systems [3]. Population fragility, however, shows that we must think more comprehensively about sharing, considering the implications at multiple levels. When components are shared, we get benefits such as increased functionality and standardization; however, that sharing comes with the cost of shared vulnerabilities. To the degree that diversity reduces the degree of sharing, diversity can be seen as something that reduces the fragility of a population. However, software diversity, in itself, does not necessarily imply a significant reduction in population fragility to the degree that there are vulnerabilities in parts that are still shared. When adding diversity, then, we have to be mindful

of whether our diversification strategy will impact whatever vulnerabilities that are of concern.

As surveys of automated software diversity show, the field is focused on the problem of changing how systems operate at a low level using randomization techniques so as to make the exploitation of vulnerabilities more difficult [20]. We believe that automated software diversity is often not diversity at all, and in fact it is a mistake to necessarily associate it with the implementation-level diversity provided by N-version programming and functional diversity. We assert that diversity should be defined as strategies that minimize population fragility. Compile-time or runtime automated approaches that would disable unneeded software components on a given host change could reduce population fragility; runtime approaches that simply randomize program behavior, however, do not. Randomizing memory layout makes it harder to perform code injection attacks; however, an attack that works against one system will work against any system that runs the same software. The key insight here is simple: making attacks harder does not reduce population fragility. The only way to change population fragility is to change the fraction of a population that is affected by a vulnerability.

While we believe the intuitions of population fragility are clear, it is not so clear to what extent the measure can help us understand real-world systems. One possible extension of our model would be to populations with non-uniform distributions of component configurations (which may require expanding our concept of fragility). Such an expanded model could then be used to directly model real-world systems. We suspect such work may lead to non-intuitive conclusions such as the Android ecosystem may be less fragile than the iOS ecosystem due to the widespread practice of phone manufacturers customizing their Android images; whether this is in fact true will depend upon the impact of vulnerabilities in shared components, particularly those distributed directly by Google.

So much research in computer security is focused on finding ways to exploit vulnerabilities and ways to mitigate or eliminate those vulnerabilities. Despite many years of effort, this arms race seems to become ever more ferocious over time. If defenders are to ever improve their position, they need strategies that will change the game. We hope that metrics focusing on a population-level view of vulnerabilities is a step on the way to changing the game to make defending systems easier than attacking them.

References

[1] S. Lipner, "The trustworthy computing security development lifecycle," in *20th Annual Computer Security Applications Conference*. IEEE, 2004, pp. 2–13.

[2] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[3] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Querterman, and B. Scheier, "Cyberinsecurity: The cost of monopoly–how the dominance of microsoft's products poses a risk to security," *Computer & Communications Industry Association Report*, 2003. [Online]. Available: http://www.ccianet.org/papers/cyberinsecurity.pdf

[4] A. Jaquith, *Security metrics*. Pearson Education, 2007.

[5] F. Freiling, I. Eusgeld, and R. Reussner, "Dependability metrics," *Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany*, 2008.

[6] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–35, 2016.

[7] L. Wang, S. Jajodia, and A. Singhal, *Network security metrics*. Springer, 2017.

[8] Á. Longueira-Romero, R. Iglesias, D. Gonzalez, and I. Garitano, "How to quantify the security level of embedded systems? a taxonomy of security metrics," in *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2020, pp. 153–158.

[9] P. Höfner, R. Khedri, and B. Möller, "Feature algebra," in *Proceedings of the 14th International Symposium on Formal Methods (FM 2006)*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 300–315.

[10] M. R. Lyu, J.-H. Chen, and A. Avizienis, "Software diversity metrics and measurements," in *1992 Proceedings. The Sixteenth Annual International Computer Software and Applications Conference*. IEEE Computer Society, 1992, pp. 69–70.

[11] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[12] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese, "Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 5, pp. 1071–1086, 2016.

[13] Q. Tong, Y. Guo, H. Hu, W. Liu, G. Cheng, and L.-s. Li, "A diversity metric based study on the correlation between diversity and security," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 10, pp. 1993–2003, 2019.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[15] Q. Zhang, "Aspect-oriented product family modeling," Ph.D. dissertation, McMaster University, Hamilton, ON, Canada, June 2013.

[16] P. Pohjalainen, "Feature oriented domain analysis expressions," in *InNordic Workshop on Model Driven Software Engineering (NW-MoDE'08), Reykjavik, Iceland*, 2008.

[17] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-; oriented reuse method with domain-; specific reference architectures," *Annals of Software Engineering*, vol. 5, no. 1, p. 143, 1998.

[18] M. L. Griss, "Software reuse architecture, process, and organization for business success," in *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*. IEEE, 1997, pp. 86–89.

[19] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating feature modeling with the rseb," in *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*. IEEE, 1998, pp. 76–85.

[20] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.