The Journal of the Southern Association for Information Systems

Volume 9 | Issue 1 Article 2

3-15-2022

Toward Remaking Software Development to Secure It

Jonathan Jenkins
Middle Georgia State University, jonathan.jenkins2@mga.edu

Recommended Citation
Jenkins, J. (2022). Toward Remaking Software Development to Secure It. The Journal of the Southern Association for Information Systems, 9, 15-37. https://doi.org/doi:10.17705/3JSIS.00020

Toward Remaking Software Development to Secure It

Cover Page Footnote
https://doi.org/doi:10.17705/3JSIS.00020

ABSTRACT

Modern software development depends on tools and techniques to represent implied information processing logic to the human engineer, relying chiefly on effortful human reasoning to best determine critical properties of the software system. Current conceptualization, visualization and contextualization of software in development amounts to a significant under-utilization of already limited development resources directed to optimization, prevention, and addressing fundamental security properties of the software system. As a step toward increasing such utilization as a basis for a global ecosystem of secure software, this work explores and evaluates an alternative representation of software source code for the sake of secure development, manifesting universal, critical properties of the system to enhance control of security-determinative factors while the bulk of the properties of the system are being determined and the costly skills of the developer are directed to the many aspects of the task.

KEYWORDS

Software Development, Secure Software

INTRODUCTION

With software systems bearing an increasing share in the provision of modern services, there is a spectrum of activities aimed at the realization of secure software systems, from the more timely mitigations of vulnerabilities during the engineering process to the "late" application of security updates to deployed systems. With a growing set of particular instances of costly security incidents (PwC, 2020) and ongoing vulnerabilities that are slowly addressed and costly to society (Veracode, 2021), responses have relied on an approach which either increases the labor and costs of developers or reactively expends resources to mitigate vulnerabilities that have already led to incidents damaging to business processes and reputation. A well established attribute of software flaws is that the cost of repair increases as the point of intervention advances further in the software engineering process, where less advantage can be made of the focused talent and more diversion of resources becomes necessary.

Although the introduction of flaws into software systems is an assumed, admitted presence in modern software engineering, the discipline has seen an evolution of alterations to software architecture, design, implementation and maintenance that support mitigation of risk and resistance to intentional attack. At the most systematic perspective, the use of layered architectures is one example of a design approach that achieves an isolation of system components from each other, enforcing a controlled interface between them and restricting unauthorized information flow. Such architectural measures set the stage for security 'from the ground up,' but do not address the creation of exploitable flaws during the implementation of the source code.

Modifications to software development tools and techniques for security purposes have largely focused on overt source code phenomena and practices tied to particular vulnerabilities. Turning to more focused interventions, manual analysis of source code for the removal of security flaws is both an effective and significantly costly addition to the engineering process, while automated security scanning tools experience practical limitations in performance and power of revelation that limit their application. Secure programming has a meaningful role in a large scale secure software effort, yet requires intentional vigilance in the observance of coding practices beyond those needed to meet software requirements.

Notably, training of software developers in secure programming techniques is a more direct attempt to address the human element in the creation of software systems, but also represents a reduction in the effectiveness of employment of the developer's human resources: the developer trains to observe practices beyond those needed for the construction of software rather than benefiting from *an adjustment to their own development paradigm and conception of the software*. Existing approaches force the

developer to apply human effort to abstractly represent the properties that are determinative of the security of the software system in development.

The modern expansion of software into the conveyance of a wider envelope of business services must motivate an "all hands on deck" exploration of fundamental alterations to the paradigm of secure software development. The exploration of such alterations is necessary in order to reveal protections that enhance the developer's efficacy without adding 'passes' to the development process, protections that mitigate vulnerabilities in a more far-reaching manner rather than leave each new attack to be addressed separately. The widely practiced reactive approach to securing software systems follows from a prioritization of functionality before risk, but can only attempt to prevent further damage within software systems already exploited, especially with defenses (e.g. "patches") that are not systematic.

The contribution, the result, of this work is a fundamental change to the development representation of the software system in order to enable the explicit manifestation of properties that have direct influence on the security of the deployed system, yet are not "visible" in raw source code. From the novel model, manifestations of the key properties are derived for use as part of the visual representation of software during development. Further, the feasibility of the novel representation and its ability to convey the corresponding augmented security information in support of development are evaluated through the creation of a prototype source code editor in Java.

The work presented here also acts as a step (a measured step in line with practical constraints of modern software engineering) toward more effective development models of software that deemphasize non-functional aspects of the system, such as labels, in favor of properties that objectively prescribe critical functionality. The new representation is realized as part of the software development process, where source code text is segmented and augmented to allow effective, explicit depiction of, and reasoning about, properties such as information flow.

The change in representation proposed here can be instructively compared to the major transition in the construction of software systems that occurred as part of the introduction of object oriented programming near the end of the 20th century. Where object oriented programming introduces the object as the new, chief structural unit of work in software (yet does not alter the work done), the proposal here preserves the functional structure of the system under development, but segments functional source code symbols to enable representation and prioritization of properties, augmenting the system representation to explicitly position the properties to inform development as part of the development tool set rather than in their traditional, abstract role requiring training and cost to employ.

The advocacy for such widely applicable software development interventions additionally reflects the responsibility for globally secure software as a *shared burden*. Since there is an unavoidable relationship of reuse and dependency between modern software components (originating within the same product, or not) and systems, the importance of bidirectional assurances of security derived from commonly employed, general protections is demonstrated.

In the remainder of this work, the most applicable work of similar approach to the present effort is reviewed, followed by coverage of the global software development context in which this work is born, an explanation of the foundational principle of properties that is to replace a pure language-based conception of software, an explanation of the feasibility of changing and augmenting the representation of software in development for security purposes, a specification of the changes made to the developer's software development environment in order to capture the properties introduced, an evaluation of the feasibility of the proposed property-based development paradigm in the form of a prototype source code editor, and finally conclusions.

RELATED WORK

There exists an established set of options for the intervention toward the security of software, both in development and as deployed, but through the best efforts of the author, no significant efforts toward the proposed degree of fundamental alterations of the development software representation for secure development are found.

However, to address the major task of ubiquitously secure software, the spectrum of practices that support engineering of secure software systems must be considered as a complementary combination rather than a set of mutually exclusive, competitive categories.

Security tools provide scanning services to identify potential vulnerabilities in software systems without continuous human intervention, yet such tools are realistically applied as assistive processes for the human developer, and may require the creation of tool-specific analysis data as a prerequisite (Wichers et al., 2020; NIST, 2020).

Manual effort by the human developer provides more intelligent and abstract security analysis of a software system, yet introduces cost and falls short of the performance of automated tools. Software scanning services such as the ample Fortify static code analyzer (MicroFocus, 2020) automate the security analysis of software as a service, dynamically analyzing executing code, and may be complemented with manual efforts to secure the system, such as penetration testing. With scaling and performance now commonly delivered in cloud environments, the developer is provided with rapid feedback on the tool's findings and able to take corrective action.

Recent developments in the security analysis of software systems have led to the employment of 'semantic' reasoning in order to identify and mitigate security vulnerabilities. This form of analysis, to be treated distinctly from syntactic and lexical analysis of code, focuses on identifying and analyzing the meaning and intent of the expressed code statements that is not expressed in the source code. Semantic analysis leads to the identification of patterns of dangerous code that are visible in, for example, whether variables are defined before use, and comparing expressed types to required types.

The addition of security analysis capabilities to the GNU compiler collection (GCC, 2010) represents a useful indicator of the penetration of such techniques into the wider creation of software. Though there can be significant impact on compilation performance, the compiler is able to scan for a set of distinct "bugs" such as double free of memory references and the "unsanitized" used of an index value.

Secure programming practices (Oracle, 2020; OWASP Project, 2020; Red Hat Inc., 2020), when already constrained attention and resources are available, are employed during software development as additive adjustments to the implementation of functional requirements, with examples found in the validation of input data and the use of security services such as encryption, authentication and access control. Though the benefits of secure practices with respect to security incidents are significant and clear, the practical adherence to proper practices is limited by a justified focus on core functionality during software development, a lack of immediate security consequences and a lack of ubiquitous training.

Security engineering addresses the mitigation of risk during the professional production of complex software products by intervening at the architectural level, where decisions can be made to build systematic protection 'into' the system with e.g. layering or code reviews, or also by applying dedicated testing for vulnerabilities. Moving toward the organizational context, risk management can be applied to identify and reduce the risk factors of the software project in ways that may reduce vulnerabilities.

As a representative example of efforts to alter the software development support structure in service of more effective development, the work in (Bragdon et al., 2010) alters the containment representation of source code, but does not adjust the representation of the code itself, allowing no opportunity to reveal the breadth of the security-relevant properties of the software system in development.

The work in (Xie et al., 2011) implements interactive support for secure programming within the integrated development environment (IDE), but relies on heuristics to detect "common" security issues in program source code. The project shares with the current work an emphasis on improving the code during development, as well as enabling useful annotations that support the developer.

In summary, there is a body of work that aims to analyze the software system in development, in source code, in order to identify code sequences which are judged to match a known vulnerability pattern. However, such measures are dependent on language-specific constructs and labels, and are tied to particular vulnerabilities. Though there is typically a set of particular categories of vulnerabilities which are prominent (OWASP Project, 2020), new attacks continue to appear as variations that present a new 'fingerprint' which does not match existing signatures in spite of potential similarities in strategy or pattern.

LANGUAGE VS. PROPERTIES

A conventional, high level representation of software, for software development purposes, is the product of an effort to represent a complex expression of logic in a manner that resembles natural language, sufficiently for the purpose of development. However, the use of high level programming languages for the expediency of human interpretation does not manifest a set of inherent properties that drive important consequences of its execution, yet are not evident to visual inspection of source code.

A question beyond the scope of this work is on the general necessity of language-bound details in the development representation of software. Although there are various practical factors that drive the creation and maintenance of distinct programming languages, a *property-based* development representation of software would be capable of an increase in efficiency through the condensing of language-specific details into common logical operations in a unified syntax. Nevertheless, rather than attempt to summarily discard widely used programming language environments on which vast running software systems depend, the approach taken here is to compromise via a representation that preserves language-bound details (keywords, syntax symbols), yet alters the structure of the presented system enough to introduce a general structure that manifests properties within the user interface and augments the display elements for the purpose of the manifestation of the security properties discussed here.

Information flow is a well-studied aspect of software source code and execution (Denning, 1976). The flow of influence between data storage locations serves as an effective model of the undesirable propagation of untrusted data and of key elements in security attacks on computer systems. The transfer of data from restricted access storage locations that are marked as sensitive to others that are less sensitive represents a clear security violation in military contexts and in others where information is controlled.

There are expressions of information flow which may be deduced within program source code, yet do not exhibit explicit transfer of information between storage locations. Such implied information flow occurs when information about a stored value **a** can be inferred through, for example, the effect of a conditionally executed statement on a separately stored value **b**. In a statement such as **if a == 0 then b = c**, the explicit information flow from storage location **c** to **b** through assignment is clear, yet conditional. An implicit flow also exists from **a** to **b**, since the storage of the value in variable **b** exposes information about the value of **a** that is checked in the conditional statement. In the model studied here, explicit information flows are accounted for, while implicit flows are viewed as a direction for future study.

Information flow is revealed as a controllable property of most programming language environments that feature programmer-managed storage, but it is not generally exposed to software developers as they work to produce software systems which rely on such flow for the useful effect of the systems. In spite of its relatively unmanifested role in the software development process, information flow models an enabling factor that is a key driver of transitions in security-relevant properties that are discussed in this work.

Supported by information flow, a complex of properties are expressed at the composition of data and logic which is described by the function interface. The definition and call of functions is one of the fundamental software constructs that enable the construction of large scale, robust software systems: the function code, bearing unenforced requirements for secure operation, is reused to process unique data of remote origin and distinct requirements for secure use. Where a function is employed, information flow facilitates the action of multiple related, security-relevant properties, to include structure, parity and extent (and the potential for more, to be studied).

The basis for the focus on the call interface as the chief locus of property action in this work is its characteristic imposition of interactions between data elements and the interactions between the invoking code and the code which is invoked. There is also a characteristic discontinuity between the intent and safety requirements of both 'sides' of the functional interface, leading to countless security vulnerabilities, and revealed here to be expressed as action via properties. Beyond that, the concept of composition of requirements is widely applicable to software constructs, opening the way for future study.

Firstly, the parity property (Table Security Relevant Properties Captured and Manifested) describes the state of symmetry between subsets of the data operated on by the invoked code. Commonly, parity is required in order for the correct and secure operation of functions which rely on a 'match' between specifiers for parameter count and the number of actual parameters passed in. Mismatches between specifiers and parameter values in calls to format functions can result in undefined and insecure program behavior. Such phenomena is famously observed in format string security vulnerabilities, which have demonstrated the capability to leak information and execute unauthorized code (OWASP, 2020;Teso, 2001).

Property	Description	Security Phenomena Captured	Practical Examples
Information Flow (explicit)	Propagation of information from source storage location to destination	Integrity violations via the transfer of information to protected data repositories, confidentiality violations via the transfer of restricted access information to unauthorized repositories	Sabotage of data, data corruption, information leak
Composition of Parity	Imposed interaction between data items via their mutual presence or absence	Potential integrity, confidentiality and availability failures due to the absence of correct, required data item for each required input	Format string vulnerability, e.g. C printf function
Composition of Structure	Imposed interaction of data with data, via a structural or topological operation	Potential integrity, confidentiality and availability failures due to the use of incorrect data to interact with data as part of structural operation	Indexing vulnerabilities, e.g. use of out of bounds index. Attacks with crafted input, e.g. parsing
Composition of Extent	Imposed interaction of magnitudes of data with data	Potential integrity, confidentiality and availability violations due to the transfer of data from source to destination, where the violations are a result of unmatched safety requirements between data and interface	Buffer overflow vulnerability, overwrite of data in memory through lack of bounds checking

Table 1: Security Relevant Properties Captured and Manifested

The structural property captures the use of incoming data by the function to impose spatially partitioning operations on data or functionality. A prominent example of the structural property in action is in the indexing of sequence variables, where a numeric (or other) data value 'selects' the particular data element to access or process. The use of unvalidated data to index a sequence data structure is a recognized pathway for multiple attacks.

An extent property is visible in the relative magnitudes of the data that is made by the function to interact. In the implementation activity of software engineering, it is the information flow that enables the potential extent interaction which is visible for analysis. The measurement of the action of information flow through extent can be fully measured only with analysis of executing software, since the true relationship between magnitudes is defined only when data is in use. Still, the incompatibility between the extents

presented by actual data and the requirements for secure reuse of the function directly describes the conditions for the endlessly applied buffer overflow attack (Aleph1, 1996).

That there are meaningful properties of the actions implied by source code is evident to cursory analyses. However, existing strategies for exposing such properties to the software engineer are not efficient or scalable, requiring for example new analyses of the same codebase, with separate development efforts aimed at addressing the consequences of software failures. Ultimately, the increase in valuable information obtained by the manifestation of properties justifies the general re-representation of source code in a property-exposing form.

REMAKING SOFTWARE DEVELOPMENT FOR SECURE DEVELOPMENT

While secure software is inarguably a critical goal for today's digitally-driven services, there are clear limitations to an approach in which each software vulnerability is analyzed to the exclusion of others and addressed with reactive, tailored defenses. As successful and necessary as a patch for a specific software vulnerability is, it is applicable to the identified flaw and does not inherently apply even to related vulnerabilities. Notably, attackers successfully renew their attacks with minor adjustments to techniques and implementations.

A more general approach to increasing the feasibility of secure software development is through a combination of re-representation of the logic to be implemented and the addition of augmented information that manifests security-relevant properties. The high level representations of program logic used in modern programming languages take the form of recognizable natural language symbols, reducing the burden of the developer's task in directly reasoning about abstract logic. However, such representations also demonstrate the amenability of program logic to re-representation, an opening to further adjustments toward a new baseline of secure software in development.

Conventional software development code representations and practices are a significant under-utilization of time, effort, and other resources, and an under-application of parallelism in terms of the overtness of software properties that are critical to the software system in production. There is a way forward to an increased utilization of the valuable human resources engaged in software development through alternative representations of the software system that 'prefer' properties rather than localized labels, exposing properties that aid the classification process of identifying vulnerable code.

The *composition* of distinct sets of source code constructs (and their implied execution effects) is a key construct which enables the effective application of a language-agnostic, property based software development methodology. The concept of composition can be applied to the most fundamental constructive technique of software development, the specification and call of external procedures, though its application is not limited to this aspect.

The definition of, and call to, reusable procedures is a fundamental enabler of the construction of complex, maintainable software systems. A trade-off of control for expediency takes place in the engineering of modern software products, in which the developer's own implementation directives are replaced with the ability to call a procedure implemented to honor a 'contract' of work done with the incoming data.

A key realization of composition is at the function interface, where arriving data carries with it a set of requirements for safe use, in terms of its extent, its parity with the remaining data items, and the imposition of structural interaction between separate data items by the function instructions. Thus, the function interface composes two potentially disjoint sets of security requirements, leading to an opportunity for the violation of the intent of the program author.

Importantly, the actions modeled by the security properties, and the attacks which are derived from action through the properties, are only truly realized during the execution of the software system. As a result,

software development with a property representation of a software system is best aimed at constraining undesirable property 'pathways' in the source code prior to program execution. The program resulting from such development disallows uncontrolled pathways through the security properties, and thus prevents the attacker from constructing security violations in terms of sequences of property state changes.

The judgements of the software developer, when informed by a view that exposes and prioritizes key properties which normally require intentional effort to reason about, benefit from a new baseline in representational accuracy of a complex system under construction. The increased accuracy is reflected in the manifestation of the properties that implement the intent of the author and determine security, rather than merely in terms of the labels of a single language system. As a practical matter, such an advancement can be realized by, for example, useful visual indicators of dangerous property disparities displayed within the development environment.

For the sake of exploration, a valid, alternative approach for the augmentation of software development practice for security purposes is the creation of a new 'special purpose' programming language environment that enforces additional requirements with the goal of reducing the likelihood of security vulnerabilities. Apart from the significant effort involved in such an endeavor and the obstacles to adoption among a context of legacy software, such a task would recreate with a new set of labels the familiar limitations of conventional programming languages in the face of security threats. An exploitation of a program implemented in such a language abuses its characteristic capabilities and restrictions, and defenses envisioned during development are traditionally expressed as alterations within the label system of the language, rather than the transformations in universal properties that truly determine the security of modern software.

In summary, a property-based representation of software for the purpose of secure development will expose and prioritize security properties that are directed by the code expressed yet not overtly manifested, and it will further augment the development task with sufficient information for the developer to curtail uncontrolled property transitions.

RE-REPRESENTING SECURITY DURING SOFTWARE DEVELOPMENT

Taking the construct of composition as a guiding principle for the support of secure software development, a property-based software development source code representation can meaningfully expose indicators of security property state changes that are implied by source code in development. The function call is a key interface at which the composition of these properties acts, and at which our novel software representation exposes this complex for the benefit of the software developer toward implementation better reflective of the process in progress.

The goal of a property-based software development representation of software must be a concise model that manifests the 'pathways' for security-relevant flows and discontinuities, but avoids obscuring the remaining properties that may be of interest to other development views. For the sake of generality, the representation of software is best formed in a property-agnostic way that assumes a granularity appropriate for the individual elements of statements which can exhibit information flow or composition. Achieving such generality is possible through the segmentation of code statements at least to the extent that information flow sources and destinations are given exclusive positioning.

Moving to the particular security properties introduced, the first property that must be represented, information flow from a source storage location to a set of others within a code scope or "block", can reveal the passage of information to unauthorized (or insufficiently privileged) carriers or repositories of that information. Unauthorized information flow represents a violation of the confidentiality principle, among the three of the classical CIA triangle. A key manifestation of information flow during software

development is, given a storage label, a visual indication of all transfers of information from the referrent of that label to other locations, including indirect flows, within the block.

Parity reflects the degree to which separate references, composed by code to interact, bear mutually present data. Parity can be indicated in the development view by the visible distinction between a data parameter string that contains a probable specifier pattern and the remaining data elements that are required to be present to 'match' the specifier parameter. To maximize the generality of the parity information displayed, the parity analysis of parameter strings is approached as an abstract pattern repetition search, using a maximum likelihood detection strategy in which the pattern most repeated is chosen as the probable repetition pattern.

Structural interaction between data entering the function can be indicated with suggestive symbols placed with the augmented parameter representation. Notably, the important indication of cross-parameter structural interaction is directional, and symbols should be sufficiently expressive for this purpose. Information flow is also directional, and can be similarly indicated within the parameter representation by symbols that illustrate the direction, source and destination of flow between function parameters.

A unifying interface concept for the manifestation of unsafe composition is the visual distinction between the elements of the composition that would exhibit similarity in a secure configuration. A standardized awareness of the presence of such interactions before the system is tested is a necessary prerequisite for informed development of secure software. To indicate the information flow between function parameters, an appropriate approach is to augment the representation of the function call element within the statement by adding a compact representation of all parameters that provides a directional indicator to indicate, for each parameter *p*, which other parameters are recipients of information flow from *p*.

Considering the property-based development representation advocated here in relation to existing approaches to the construction of secure software, the present approach either advantageously provides support earlier in the lifecycle of the software system, or reduces the reasoning and classification burden placed on the developer seeking to mitigate vulnerabilities. Further, this work advances efforts toward an entirely general representation of security-relevant phenomena during development, which is applicable to essentially all software systems to be built.

Rather than scanning for "top 10" vulnerabilities or variations of studied vulnerabilities, the properties manifested in the present work can capture precursors to entire categories of attacks, and developers benefit from their overt representation as part of a holistic view during the implementation process. The actual mitigation of vulnerabilities is left to the human developer, given that alterations to the system source code are required. Here, a comparative analysis of property based development relative to existing work is conducted.

In comparison to the present work, manual effort by the human developer provides more intelligent and abstract security analysis of a software system, and security testing of deployed software systems can clearly mitigate vulnerabilities, yet the resources consumed by adding developer effort are significant, both during development and after deployment. These forms of alterations to the engineering process can contribute to the reduction of vulnerabilities, but remain part of an intentional effort that must be employed uniquely to each project: it cannot be employed as a new 'baseline' of security support apart from the development effort.

Relative to manual intervention during or after development activities, a property-based model of development acts as a transfer of cost from human resources used for representational effort and classification to an 'up front' cost for the implementation and adoption of the model within development tools and practices. However, such a transfer is not complete, since the vulnerability classification task, though reduced in degree, still requires an intelligent assessment of risk based on the unique presentation of property effects of the present software system. The property-based development approach

nevertheless provides a 'baseline' of increased security, since the developer's challenging classification task begins with augmented support.

Risk management offers principles and software project practices that can objectively identify project risks earlier, and may 'save' a software project. There exists in the discipline of software engineering an interest in positioning risk management as an early, guiding set of processes to reduce project costs and support success (Boehm, 1991). Risk management, as applied to software engineering, aims to coherently and precisely identify key project risk factors, preventing them from becoming specified and implemented to build threats to success into the product. One example of such a risk is in the case of faulty requirements admitting the use of insecure software features.

Although project management level intervention is not attempted by the present work, risk management can be considered complementary to property-based development. While risk management serves to prevent unknown risks (including security threats) from impacting on the project's success at an organizational level, intervening in the implementation activity (as is advocated here) instead prioritizes the developer's role in the most effective and efficient use of their human resources in reasoning with the security properties of the complex system in development and avoiding the introduction of risk.

Software testing can be applied during the development in a way that supports the security of the produced system, but can also be applied in dedicated security testing (Sommerville, 2016). Software testing is able to increase software quality and prevent flaws from remaining in software systems to be exploited by attackers, yet adds costly activities to the development process as quality increases, and is subject to all limitations of the software engineering process. After all, an accepted phenomena in software engineering is the increase in cost necessary to achieve highly dependable software systems that resist attack.

Achieving secure software systems by 'patching' systems *after deployment* enjoys a perverse benefit of the information about how a vulnerability or flaw is exploited, yet clearly does not represent a development intervention, does not minimize significant potential costs of addressing damage to business processes, reputation and services, and does not ease the complex classification and mitigation task of the human developer. On the other hand, the property based approach aims to intervene during the development process, and provide a more secure 'starting point' in development at the current view of the code.

Static security analysis of source code (Stefanović, 2020) is a major, modern approach to producing secure software, and can be considered complementary to the work proposed here, in that such analysis places the focus on the software system in its development activity. Static analysis involves the manual or automated processing of the software source code with the goal of revealing security vulnerabilities. The use of software tools for static analysis of source code (NIST, 2020 ;Wichers et al., 2020) has achieved varying levels of performance, and characteristic inclusion of false positives in warnings to the developer (Scanlon, 2018). Due to the various limitations of scanning tools, developers are not likely to extract the optimal benefit from them (Jamil, ben Othmane, Valani, AbdelKhalek and Tek, 2020; Thomas, Tabassum, Chu and Lipford, 2018).

Software security scanning services such as Fortify static code analyzer (MicroFocus, 2020) and (Veracode, 2020) provide automated, external analysis of software systems, identifying code sequences that exposes the system to risk and threat. With the use of such tools come the limitations of external analysis which can 'see' only what is evident in code, missing semantic details and any benefits of alternative developer-facing development representations of the software system.

Although static analysis can be performed with automated tools that execute at machine speed as often as needed, the breadth of the vulnerabilities that can be detected in an automated scan is limited to the capabilities of machine (not human) intelligence and the representational power of the source code,

resulting in tools typically being applied in a supporting role for security analysts working to locate and mitigate flaws.

Static analysis does not change the nature of the support available to the developer *as the source code is developed*, and does not reveal or expose the key fundamental properties that influence the security of the developed system, properties advocated here as central to the way forward in tool-supported, secure software development.

Thus, the limitations of static code analysis in comparison to the present work are categorical: the model proposed here directly alters the development representation of software to expose security-critical properties and augments the representation with development support to reason with these properties, while static analysis analyzes the source code 'as is', revealing problems that are evident to software based intelligence. The recognition of vulnerable code patterns by static analysis is dependent upon an accumulated, costly experience with past security violations that are constructed from effects of the properties discussed in this work.

Admitting the common approach of intervention during development in (Xie et al., 2011) and the current work, the interactive static code security analysis enabled by the ASIDE project is an important point of comparison. The project implements developer support for adaptable awareness of common security problems in the development environment tool, realized in an Eclipse plugin. An example of the application of ASIDE to a security vulnerability is the automatic recognition of the absence of validation for an input data item: the tool generates applicable Java source code, using standardized libraries, which can be added by the developer.

Although such interactive, in-development security refactoring clearly is capable of identifying code of concern and facilitating mitigation, the approach, further developed in (Zhu, 2014), is subject to false positive warnings that would be expected from a software-based classification capability, and does not fundamentally alter the representation of the software system toward secure systems. The present work does not attempt software-based classification, but advances a model of software that is properly representative of security-critical properties reasoned with during development, in order to increase the effectiveness of human developer classification, where true security-expertise resides.

As reflected in the plugin implemented in the work of Zhu et al., development tools such as Eclipse demonstrate not only the current state of support for general customizations of the development process, but also the amenability of development frameworks and tools to augmentation with security-relevant functional support, on which the practicability of the present work relies.

Secure programming (Bishop, 2006) is an approach to the production of secure software systems that is more directly similar to the present work than static code analysis, in that the intervention is applied to the development paradigm, rather than through the application of automatic analysis. The use of proper input validation to sanitize data originating from untrusted sources is one example of a programming practice that mitigates vulnerabilities in which input interfaces are exploited.

While secure programming can be taught to developers in training and employed to significant benefit by professional engineers to prevent well-known categories of vulnerabilities, ongoing training, resources and vigilance beyond the fundamental efforts of constructing working software are necessary to realize the benefits. In comparison to the present work, secure programming represents an allocation of tasks of arbitrary size to the developer, requiring the ongoing application of patterns to programming scenarios that present uniquely.

	Costs	Software Lifecycle	Developer Classification Effort
Software Testing	Development activity resource consumption	During development	Developer classification of incorrect or faulty code
Patching	Software maintenance costs, security testing, organizational incident response	After deployment	Developer classification of insecure code
Code analysis	Ongoing tool costs, Developer verification	During development	Partially automated classification, developer effort for confirmation
Secure Programming	Ongoing Training, Developer resources	During development	Developer classification of insecure code
Property-Based Development	Initial development tool implementation and adoption	During development	Significantly automated

Table 2: Interventions for Software Security

In context, the present model prioritizes properties which increase coverage of security vulnerabilities rather than requiring an intentional, abstract and arbitrarily complex classification process, the scale of the classification can be reduced and uniform adjustments applied to mitigate vulnerabilities (see Interventions for Software Security and Interventions for Software Security).

As depicted for key security vulnerability categories in table Table 3: Comparison of Developer Action, By Vulnerability Category, a property-based development model of software enables the transformation of the developer action from, at best, assisted identification with developer manual mitigation to significantly automated identification with supported mitigations that apply to entire categories of vulnerabilities.

	Secure Programming	Static Code Analysis	Property-Based Development
Sabotage of data, data corruption, information leak	Intentional developer training and practice for removal	Scanning for explicit flows with human intervention for each flow	Automated flow identification, developer mitigation (e.g. disabling of flows not specified in requirements)
Format string vulnerability, e.g. C printf function	Developer training and vigilance to avoid dangerous dual channel functions	Scanning for function usage, with human intervention for verification of vulnerability	Automated identification of parity risk, developer mitigation (e.g. forced parity)
Indexing, parsing vulnerabilities, e.g. use of out of bounds index or crafted input	Developer training and practice to avoid use of unchecked index values or validate index values	Scanning for source-code visible, unsafe index uses	Automated identification of indexing relationship, developer mitigation
Buffer overflow vulnerability, memory overwrite	Developer training and practice to implement bounds checking	Language-based pattern scanning , developer verification and unique mitigation per instance	Automated identification of information flow and extent relationship, developer mitigation

Table 3: Comparison of Developer Action, By Vulnerability Category

The feasibility of a property-based model of development is supported by potential implementations in the context of any programming language, given the universal nature of the security properties given priority in this work. Information flow, for instance, usefully models dangerous flows of influence between data, regardless of the language system used to direct the uses of the data in variables. The properties directed at the site of the invocation of a function remain in force independently of the data types, calling conventions or parameter passing in place for the programming environment.

With regard to the adoption of property based development, mass application is possible via the bearing of a one-time cost of implementation of the model within community-developed software development tools such as Eclipse, as demonstrated by the plugin additions described in (Zhu, 2014). Since the properties are inherent in the useful action of all modern software, the need to support unique visualizations or representations for each security concern is avoided, requiring only a useful visualization of each property of interest and potential indicators for a limited number of function parameters.

The use of a property-agnostic visual arrangement can be tuned to preserve the overt, language and label-specific details of program statements to the degree necessary to maintain development recognition to the prevailing needs of the state of the discipline, and the prioritization of properties does not create significant compatibility issues with structure in programming languages that support segment-able code statements. Since the literal source code contents are preserved, computation of property information is largely a one-time cost, and the source code representation can be augmented with visually concise indicators, no significant degradative impact on the speed of development is expected.

In summary, the property-based development representation does not force the developer to train to expose to reasoning key properties that influence software security, but faithfully positions such properties in view during the process of their direction by the developer. Without direct representation of properties during development, developers must unnecessarily train and exert effort to manifest them, consuming project resources at a disadvantage to avoid the dangerous direction of such properties, leading to vulnerabilities.

A PROOF-OF-CONCEPT PROTOTYPE PROPERTY-BASED SOFTWARE DEVELOPMENT SOURCE CODE EDITOR

For the evaluation of the feasibility and efficacy of a property-based, security augmenting software representation, a proof of concept, prototype software development source code editor, JPSCE, is implemented in the modern JavaFX graphical framework, for use with the Java programming language. Since the properties and constructs studied in this work are applicable to essentially all modern software development approaches, the possibility of the addition of support for other programming languages follows from the demonstration here, but is considered a secondary focus here and a topic for potential future work.

Importantly, as an early evaluative measure for a novel *model* of software, JPSCE is not (and cannot be) an assessment of property based development in practical use or of a secure development software tool, the former being an undertaking that can be envisioned only in future work and the latter out of scope.

Fundamental software source code editing functionality is supported in the prototype, to include the addition, transformation and deletion of source code lines, whether the modification of individual statement elements or entire source code lines. Editing of individual source code line elements, possible via the user of a separate editable text field per element, requires the use of an update button functionality to regenerate the full code element representation to take the modifications into account, and a save button writes the current status of the code to long term storage. Bulk editing of an entire source code line is possible with an editable text field at the horizontal end of each ’row’ of code elements which construct the code statement. A selection of screen shots of JPSCE in operation is presented here in figures 1-5 in an appendix.

JPSCE places source code statement elements in a grid arrangement in order to expose properties in a manner that is both orderly, and agnostic to properties of interest to the developer. Information flow is an example property which requires the separation of statement elements such as operators and storage labels. The scale of subdivision of traditional code statements must be agnostic to language and property.

The exposure of information flow within code blocks is made possible on demand, relative to a single flow source of choice, via the use of event-driven programming and clickable element boxes. A click applied to a code element for a statement within a block, if the code element is a potential information flow source (a storage location label), triggers an updated colored indication of the other storage labels that receive information flow from the source.

The representation of the method call is an abstract representation that includes not only the original, editable call statement code, but a representation of the composition interaction effects of the parameters to the call, and also a hide-able, contained, scrollable display of the function implementation (if such

implementation is accessible). The augmentations to the source code call are placed with the element holding the textual name of the called function.

With the call representation constructed as a vertical box, the upper most container is reserved for the definition that is matched to the current statement's call, if such a definition is directly accessible in the present source code file (as it is tied to the questionable availability of outside source code, processing remote definitions is left for future study). The method definition, held within a fixed size, scrollable window, can be shown or hidden with double click actions applied to the function name element in the source code line.

The container second from the top in the function name element representation includes visual representations of each actual parameter, with augmented indicators for the display of security-relevant properties. Finally, the vertically lowest element in the three container method name representation displays the method name, implemented to receive double click events for toggle of visibility of any locally available definition.

Within the second container, dedicated to parameter representation, the visualization of parity is implemented as color difference within the parameter that is concluded via analysis to most probably contain a specifier pattern that imposes parity in the call. Structural interaction between parameters is indicated with common indexing symbols, square brackets placed around the parameter used to index with the proposed sequence that is indexed to the left of the brackets.

Given that conventional software development, for various practical reasons previously mentioned, relies on a set of competing source code language representations, a lack of an objective, property-focused representation forces the initial analysis of language-based elements in order to reveal meaningful property-based effects as part of the development display. As discussed, the argued compromise between support for modern programming languages and efforts to generalize secure development requires that JPSCE preserve the target language string content, yet change the granularity and layout of code elements to support property-based representation.

As a demonstration of the feasibility of a property-based mode of source code development, JPSCE implements features that expose information flow and security-relevant aspects of source code interface constructs. Information flow is exposed in the characteristic flow of data from data sources (e.g. Java reference variables) in source code, and can be revealed on demand by user action upon a particular code element that is a potential information flow source. Information flow is currently implemented within the scope of code blocks, which indirectly reflects information flow interactions between data sources entering the method and data storage locations processed within the method body.

In line with the goals of the presented theory and model of software in this work, JPSCE does not attempt security scanning of existing source code, a task implemented by numerous tools referenced, but instead implements a novel representation of the software system in development that is representative of key properties that influence its security, and augments the representation with information that manifests such properties in a way that enables informed, holistic development with improved security properties.

While the property-based representation demonstrated here enables a baseline of increased attention and awareness of security-relevant factors, the large landscape of existing programming language environments that are used to express today's software systems must be acknowledged. Further, in the short term, the coalescence of programming languages toward a unified, domain independent option is not realistic. Thus, this demonstration does not discard information locked to programming language constructs and labels, preserving it within the flexible, element-based representation, though the expendability of unique language details is a worthy matter for future discussion.

JPSCE embodies the model and focus of a property-based view of software in development, demonstrating the capability to expose entirely general information that nevertheless allows informed

development, revealing key interactions that act as 'pathways' to security vulnerabilities, independent of particular variations of attack that appear in concrete code patterns.

CONCLUSION

In this work, a novel model for the representation of software systems under construction has been presented, and beyond that an evaluation of the model for the support of secure software development. A software system in development has been revealed to be amenable to representation as a sequence of implied information flows, acting through the composition of security requirements and expression of particular properties.

A practical demonstration of the fundamental capabilities enabled by the proposed representation and augmented development view has been presented. This development prototype demonstrates the feasibility of the application of the novel representation toward property-based development supported by awareness of properties which underlie all software systems. Such a property-based development process reveals the power of key security properties, information flow as a representative, well-understood example, to uniquely expose information that indicates the possibility of security vulnerabilities.

The composition of software based action with data, and the properties that are expressed at the call interfaces that direct such composition, have been revealed as key points at which to intervene during development to mitigate the introduction of vulnerabilities, if the full scope of relevant information is provided to the developer.

Left to future work are considerations of the expansion of the augmented development process to other programming environments, and the exploration of additional properties that may have more to offer a new standard of assisted, secure software development. The detection and manifestation of implicit information flows are another avenue for future exploration.

REFERENCES

1. Aleph1. (1996) Smashing the stack for fun and profit. Phrack 7, 49 https://phrack.org/issues/49/14.html.
2. Bragdon, A., Reiss, S., Zeleznik, R., Karamuri S. , Cheung, W., Kaplan, J. , Coleman,C., Adeputra,F. and Laviola, J. J. (2010) Code bubbles: Rethinking the user interface paradigm of integrated development environments. In Proceedings of the 32nd International Conference on Software Engineering, 455-464, doi: 10.1145/1806799.1806866
3. Bishop, M. and Orvis, B.J. (2006) A clinic to teach good programming practices. In Proceedings from the tenth colloquium on information systems security education, 6, 168–174
4. Boehm, B. (1991) Software Risk Management: Principles and Practices. IEEE Software, 8, 1, 32-41, doi:10.1109/52.62930
5. Denning, D. E. (1976) A lattice model of secure information flow, Communications of the ACM, 19, 5, 236-243. https://doi.org/10.1145/360051.360056
6. GCC (2010) Options that control static analysis. https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html
7. Red Hat Inc. (2020) Secure coding. https://developers.redhat.com/topics/secure-coding

8. Jamil A., ben Othmane, L., Valani, A., AbdelKhalek, M., and Tek, A.. (2020) The current practices of changing secure software: An empirical study. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, 1566-1575, doi: 10.1145/3341105.3373922.
9. MicroFocus (2020) Fortify static code analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview
10. NIST (2020) Source code security analyzers - samate https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.
11. Oracle (2020) Secure coding guidelines for Java SE, https://oracle.com/java/technologies/javase/seccodeguide.html
12. OWASP (2020) Format string software attack. https://owasp.org/www-community/attacks/Format_string_attack
13. Open Web Application Security Project (2020) Owasp secure coding practices-quick reference guide. https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/migrated content
14. OWASP Project (2020) Owasp top 10. https://www.owasp.org/www-project-top-ten
15. PwC (2020) Global economic crime and fraud survey 2020. http://www.pwc.com/gx/en/services/forensics/economic-crime-survey.html.
16. Sommerville, I. (2016) Software Engineering, 10, Pearson, Hoboken
17. Scanlon, T. (2018) 10 Types of Application Security Testing Tools: When and How to Use Them. https://insights.sei.cmu.edu/sei blog/2018/07/10-types-of-application-security-testing-tools-when-and-how-to-use-them.html
18. Scut/Team Teso. (2001) Exploiting format string vulnerabilities. https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf
19. Stefanović, D., Nikolić, D., Dakic, D., Spasojević, I. and Ristic, S. (2020) Static Code Analysis Tools: A Systematic Literature Review. Proceedings of the 31st DAAAM International Symposium. Published by DAAAM International, ISBN 978-3-902734-xx-x, ISSN 1726-9679, Vienna, 10.2507/31st.daaam.proceedings.078.
20. Thomas, T.W., Tabassum, M., Chu, B., and Lipford, H.. (2018) Security during application development: An application security expert perspective. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, 4, 1-12. doi:10.1145/3173754.3173836.
21. Veracode (2020) Application analysis. https://www.veracode.com/products/application-analysis.
22. Veracode (2021) State of Software Security v11. https://info.veracode.com/report-state-of-software-security-volume-11.html
23. Wichers, D., itamarlavender, will obrien, Eitan Worcel, Prabhu Subramanian, kingthorin, coad-aflorin, hblankenship, GovorovViva64, pfhorman, GouveaHeitor, Clint Gibler, DSotnikov, Abraham, A., and Rathaus, N. (2020) Source code analysis tools https://owasp.org/www-community/Source_Code_Analysis_Tools.
24. Xie, J., Chu, B. and Lipford, H.R. (2011) Idea: Interactive support for secure software development. In Proceedings of the Third international conference on Engineering secure software and systems, ESSoS'11, 6542, 02, 248-255, doi: 10.1007/978-3-642-19125-1 19
25. Zhu, J., Xie, J., Lipford, H.R. and Chu B. (2014) Supporting secure programming in web applications through interactive static analysis, Journal of Advanced Research, 5, 4, 449-462, ISSN 2090-1232, https://doi.org/10.1016/j.jare.2013.11.006

Appendix

FIGURES

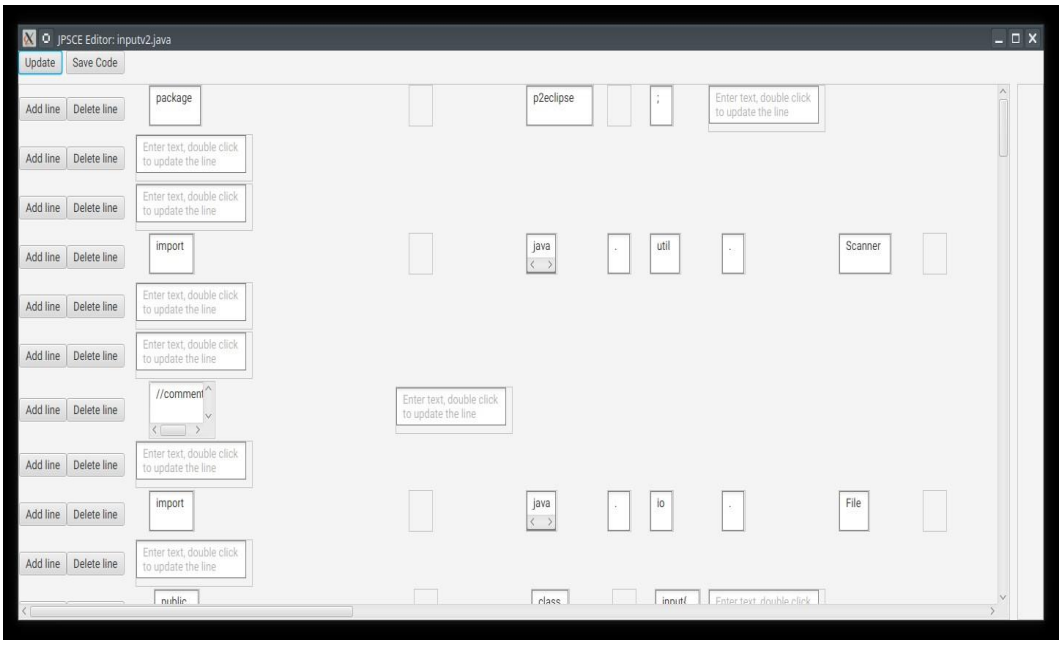

Figure 1: JPSCE User Interface

Figure 2: Information Flow Indicators

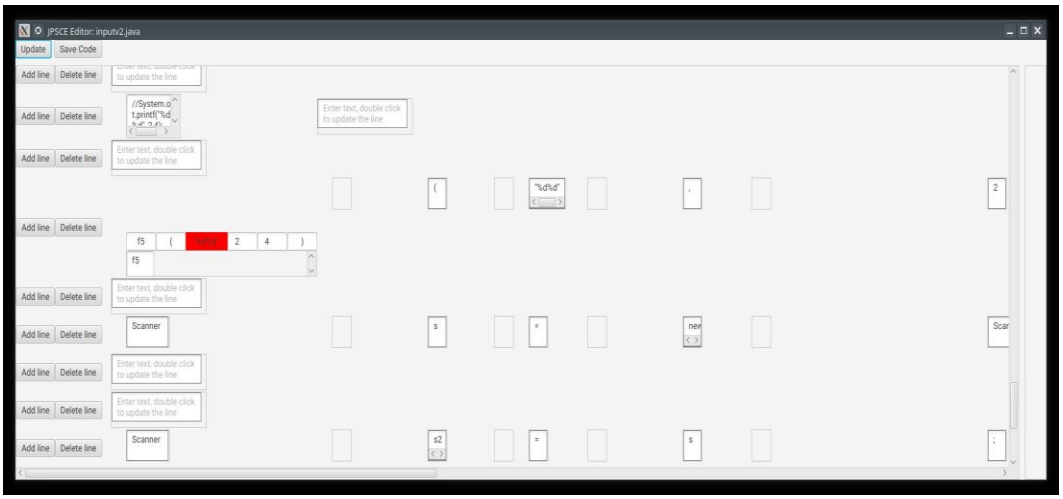

Figure 3: Function Call Parameter Parity Visualization

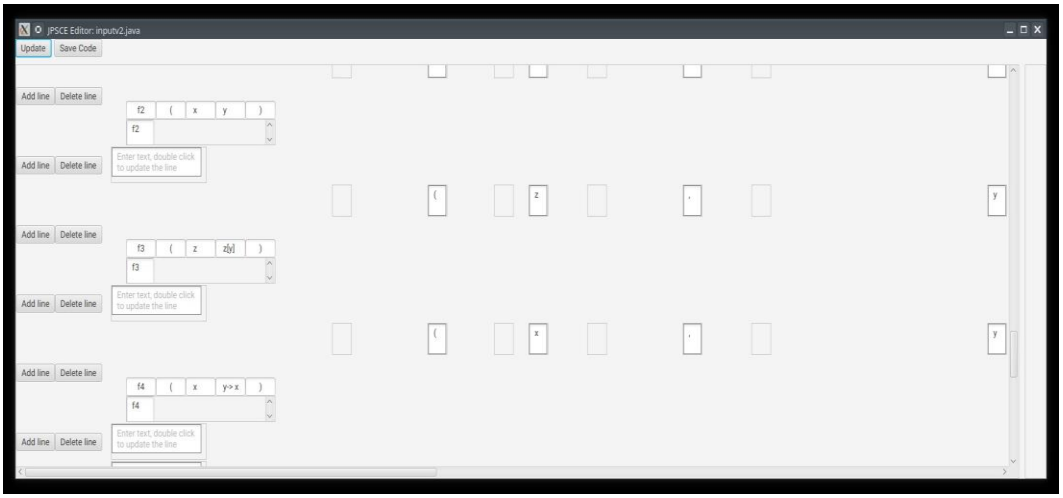

Figure 4: Function Call Parameter Structural Interaction Indicator

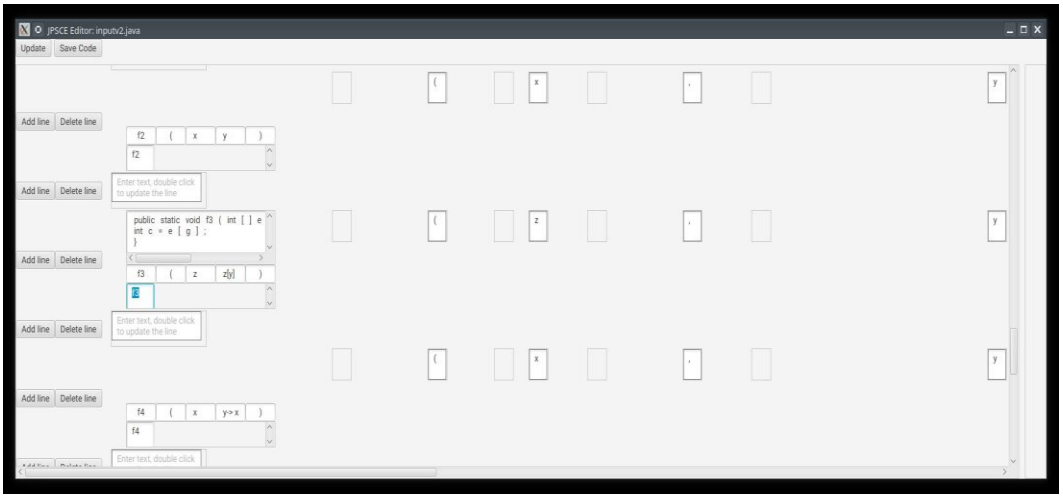

Figure 5: Toggled Visibility of Available Definition for Called Function

Sample Java Input File

```
import java.util.Scanner ;


//comment

import java.io.File ;

public class input{

private static String key ;
private static String imgfn ;


public static void f ( int a , int b ) {
int c = a ;
b = c ;
}

public static void f2 ( int ... a ) {
int c = a [ 0 ] ;
int b = c ;
int d = 2 ;
int e = b ;
int j = c ;
}

public static void f3 ( int [ ] e , int g ) {
int c = e [ g ] ;
}

public static void f4 ( int a , int b ) {

a = b ;
}
```

```
public static void f5 ( String fmt , int ... a ) {
;
}

 /**
  * The main method is only needed for the IDE with limited
  * JavaFX support. Not needed for running from the command line.
  */
public static void main ( String [ ] args ) {

 //read in a key password

int x = 2 , y = 3 ;
int [ ] z = new int [ 4 ] ;

f2 ( x , y ) ;

f3 ( z , y ) ;

f4 ( x , y ) ;


//System.out.printf("%d %d", 2,4);

f5 ( "%d%d" , 2 , 4 ) ;

Scanner s = new Scanner ( System.in ) ;

Scanner s2 = s ;

System.out.println( "Enter an image file name on a line" ) ;
imgfn = s.nextLine ( ) ;
}
}
```