Requirements for a Specification Language for Data and Information Flow, and A Literature Review of Analytical and Constructive Research on the Java Native Interface

Technical Report TUD-CS-2017-0025, January 2017

Ben Hermann[2], Ximeng Li[1], Heiko Mantel[1], Mira Mezini[2], Markus Tasch[1], and Florian Wendel

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling and Analysis of Information Systems[1] and Software Technology Group[2]

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

CRISP
Center for Research in Security and Privacy

1 Requirements for A Data and Information Flow Specification Language

1.1 Introduction

One of the major themes of the CRISP project is to realize the "security-at-large" paradigm in the engineering of complex, component-based software systems. To this end, one needs to articulate the security requirements at not only the intra-component level but also the inter-component level, in software systems. We are particularly concerned with security requirements in terms of data and information flow.

A specification language is needed for the designer of IT systems to articulate such requirements. We envision that the specification language has two main uses:

1. specifying information flow *requirements* – from which domains to which domains information flow is allowed
2. specifying the *assumptions or output* of an information flow analysis – from which entity to which entity information may actually flow

For either kind of potential usage, the specification language also needs to be able to deal with system resources, since the security analysis is expected to cross the boundary between applications and libraries.

In the context of CRISP, we envision a resource-aware requirement specification *spec* for the flow of data and information in the overall system including a user application and the libraries used by it. There are two potential approaches to the verification that the overall system satisfies *spec*. In the first approach, one can verify that the application satisfies a requirement specification $spec'$ via information flow analysis such as using an elaborate variant of security type system. In this case, $spec'$ needs to be consistent with the assumptions about what flows can occur in the environment, as captured in the asumption/output language. One can then check whether *spec* is consistent with $spec'$ given the data and information flows derived about the libraries and represented again in the assumption/output language. In the second approach, one derives the potential data and information flows in both the application and the libraries, represented in the output language. One then combines the flows derived for the two parts into overall data and information flows throughout the system. Such overall flows can then be checked against *spec*.

The requirements on the specification language itself needs to be clarified for the appropriate design and development of the language. To this end, we carry out an extensive literature survey on existing specification languages for similar purposes, and present the results of this survey in Section 1.2. We then analyze the desirability of the existing features in the context of CRISP in Section 1.3. On this basis, we articulate the requirements on our specification language to be developed in Section 1.4.

1.2 Specification Languages for Information Flow Requirements

Decentralized Information Flow Policies

In decentralized information flow policies, information is owned by mutually distrustful security principals. The flow of information is regulated by policies that combine the policies of all the owners. For confidentiality, the policy of each owner further clarifies all the allowed readers that can have direct, indirect, or implicit access of the information protected by the policy. Information can flow from A to B if the policy of B has more owners, each allowing less readers than the policy of A does. Such a constraint can be circumvented by explicitly using declassification operations that relax the security requirement.

This model for specifying and checking information flow policies is called the Decentralized Label Model and is implemented for the Java language, resulting in the Jif (Java Information Flow) language and compiler [43]. In Jif, the policies are typically embedded into program code by using security labels for variables, fields, parameters and classes. The Jif language has been extended for the implementation of secure voting systems [19], Email systems [31], and distributed systems in general [39].

The correctness of the Jif compiler rests in informal arguments on what kind of information flows exist in a program, by applying the kind of intuition in the work of Denning and Denning [23] to the rich language features of Java. There exist a few theoretical developments building on the core Decentralized Label Model. These developments are carried out for small languages for research purposes, and usually include proofs of noninterference properties [63, 18, 57].

A different model and platform for Decentralized Information Flow Control is Aeolus [17], primarily implemented for Java in runtime libraries. In Aeolus, information flow labels are sets of tags. Each tag represents a particular kind of information. The labels are used for coarse-grained protection, e.g., of threads and of files. Threads and methods execute on behalf of security principals. As in the Decentralized Label Model, the principals can form a principal hierarchy via an "acts-for" relationship. However, the delegation between principals can be done in a finer-grained manner, by granting authority only for particular tags. The revocation of authority is supported. A method can be called with an "authority closure", allowing the local exercise of authority to remove local contamination of information.

Yet another label model for Decentralized Information Flow control is Disjunction Category Labels (DC labels [54]). DC labels are conjunctions and disjunctions of security principals. As is the case for the Decentralized Label Model and Aeolus, the labels form a security lattice, that is modulated by delegation between principals. We refrain from elaborating further on the details.

Flow Policies for Operating Systems

For operating systems, entities subject to protection are threads, processes, users, files, network devices, registers, etc. [61, 58, 34]. The protection comes at a lower level, and is most of the time targeting coarser-grained objects (with exceptions such as registers), compared with information flow control in user applications.

HiStar [61] is an operating system that keeps track of information flow. It uses *category-based* information flow labels attached to the aforementioned objects. Each label is a set of secrecy categories and integrity categories. Each category can be roughly understood as the trust of a principal in the secrecy/integrity of the protected object. Information can flow from one label to another if the former contains less secrecy categories, and more integrity categories than the latter does – the opposite to how principals are compared in the "relabelling" relation of the Decentralized Label Model.

Flume [34] is a "user-level reference monitor" that keeps track of information flow between Linux processes and files. The information flow labels are also category-based – in fact HiStar borrows from Flume in terms of the label model used [61].

Asbestos [58] is an information-flow-aware operating system that inspired HiStar. It adopted a more complex label model than that of Flume and HiStar. Objects are referenced using handles that correspond in some sense to the categories of Flume and HiStar labels. A label is a function from handles to priviledge levels $[\star, 0, 1, 2, 3]$ (where $\star$ is the most priviledged level and 3 the least). The Asbestos policies regulate message-passing style operations (such as inter-process communication), while the HiStar policies regulate shared memory accesses (such as the transfer of control into a different address space).

None of the aforementioned work delivers protection in the style of a noninterference property.

Micro-policies

Micro-policies [2] apply at the level of machine instructions. Information flow labels from an arbitrary lattice are attached to operands of the instructions and the program counter. The policy is specified as inference rules for a dynamic information flow monitor for an abstract machine, and refined down to a transfer function and a monitoring service for a concrete machine. In [3], the approach is extended to

address not only information flow problems, but also trace-based properties such as compartmentalization, control flow integrity, etc. at the machine-level. At the current stage, extensive formalization and refinement proofs exist for the approach for conceptual machines.

Flow Policies for Distributed Systems

Two notable systems that enforce information flow policies for distributed systems are DStar [62] and Fabric [39]. Both systems build on previous ones that used a similar model of information flow labels to achieve information flow control for monolithic machines. In the case of DStar, labels from HiStar [61] are adopted. In the case of Fabric, the Decentralized Label Model [18] is the basis. Both DStar and Fabric come with non-trivial implementations. Neither comes with noninterference guarantees.

Jeeves

Jeeves [60] is a functional language that helps protect user privacy by guaranteeing noninterference for the confidentiality of data. A programmer can use sensitive values, policies, and context values to specify conditional information flow requirements. A sensitive value is of the form $\langle v_\bot | v_\top \rangle_\ell$, where $v_\bot$ is the data value to be exposed in the public view of the sensitive value, while $v_\top$ is the data value to be exposed in the confidential view of the sensitive value. Which view is needed depends on the "level variable" ℓ, that can be conditioned on the context values. The enforcement method is close to multi-faceted execution [8]. Jeeves is implemented by translation to a policy-enforcing constraint lambda calculus, which is in turn embedded in Scala.

Information Flow Specification using Logical Formulas

The approach proposed by Andrews and Reitman early in the 80s in [7] allows the programmer to use Hoare-logic-like preconditions and postconditions that can contain comparisons of the security domains of variables. An example is $\{\underline{x} \leq \underline{y}\}\, S\, \{Q\}$, saying that the security domain of the variable x is dominated by the security domain of the variable y before the program statement S. In [7], an axiomatic semantics is developed to construct "flow proofs" for programs written in a simple theoretical language. The "flow proofs" need to be constructed according to the Hoare-like rules of the axiomatic semantics, with no proven noninterference guarantees.

Darvas, Hähnle, and Sands [21] propose the use of Dynamic Logic [30] for the specification of information flow requirements. In Dynamic Logic, the formula $\langle p \rangle \phi$ expresses that ϕ holds in any state in which the program p terminates. Consider the example policy $\forall l.\exists r.\forall h.\langle p \rangle r \doteq l$ used in [21]. It says that irrespective of the potential values of the confidential variable h that the program p is started with, p terminates with the value of the variable r equal to that of the initial value of the low variable l. This indirectly says that the initial values of h do not interfere with the final values of r, or there is no information flow from h to r. The approach is showcased by proving security and insecurity of JAVA CARD programs using the KeY tool [1].

Amtoft et al. [6, 4] propose relational assertions that can be used in the preconditions and postconditions of a Hoare-like logic. For instance, the assertion $b \Rightarrow x\ltimes$ states that if the condition b holds, then the value of x should be the same in a pair of executions differing in confidential data – thus x is guaranteed not to contain confidential information at the program point where the assertion is used. The proof system for the Hoare-like logic is sound with respect to termination-insensitive noninterference. This approach is implemented for SPARK [53] — a practically used subset of Ada with no pointers or heap-based data. SMT solvers are used for the minimization of the assertion formulas, and proof obligations establishing that preconditions are related properly to postconditions are emitted for processing in the Coq proof assistant [55].

Nanevski et al propose Relational Hoare Type Theory (RHTT [44]). As the name suggests, there are ingredients similar to the relational assertions of [4] that can be used for stating that variables do not contain confidential information. More precisely, the preconditions and postconditions are dependent types, with the latter potentially containing such relational ingredients. Via the Curry-Howard corre-

spondence [46], types can be "read" as logical propositions – the reason that it is reasonable for this approach to be classified under "information flow specification using logical formulas". In more detail, the type STsec $A\ (p,q)$ can be understood as a specification for a heap-manipulating program with return type A, precondition p and postcondition q. Here, the postcondition q "relates the output values, input heaps and output heaps of any two terminating executions" [44]. Using RHTT, one can specify both information flow policies and access control policies. For information flow security, the type system guarantees termination-insensitive noninterference, which allows non-termination, but not the values of public variables, to leak confidential information.

It is worth mentioning that relational Hoare logic (followed by [6, 4, 44]) was first introduced by Benton in [12], where a simple WHILE language was dealt with, and secure information flow was examined as one of the application areas of the logic proposed.

Temporal logics can be used to specify information flow requirements. In [20], the logics HyperLTL and HyperCTL* are proposed for specifying hyper-properties, such as information flow properties. HyperLTL supports interleaved universal and existential quantification over traces in the beginning of each formula. An example from [20] is the formula $\forall\pi_1.\forall\pi_2.\exists\pi_3.\varphi$ that says: for all traces π_1 and π_2, there exist a trace π_3 such that φ is satisfied over the three traces. The formula φ can contain the standard modalities of LTL, the standard connectives of propositional logic, and atomic propositions, which can implicitly capture a notion of security domains of the states along the traces. This formula can be used to express trace closure properties (e.g., [42]). The logic HyperCTL*, on the other hand, can be used to address observers that can discern different branching behaviors.

In [9], Epistemic Temporal Logic [29] is used to express information flow policies. A distinguishing feature of Epistemic Temporal Logic is the epistemic connectives K and L. Given a formula ϕ, the formula $K\,\phi$ says that an agent knows ϕ because ϕ holds in *all* states consistent with the agent's observations so far. On the other hand, the formula $L\,\phi$ says that ϕ holds in some of such states. Being concerned with several alternative states (possible worlds) enables one to express security properties that are inherently about multiple runs, such as noninterference.

In [24], the authors propose the temporal logic SecLTL to express conditional declassification policies. A *hide* operator is introduced, allowing one to state that the observations of a system are not affected by the initial values of secrets before a condition starts to hold. Semantically, the hide operator is interpreted by considering all alternative paths of the target system started with different secret values, and requiring that the observable parts of these paths coincide, until the condition starts to hold on the current path.

Temporal logics usually specify the security policy and security property at the same time. The most natural verification method for a temporal logic specification is model checking. However, battling against state space explosion is still an ongoing research area in software model checking [32]. The verification of security properties is only more difficult than the verification of safety properties typically addressed in software model checking. Thus the temporal logic specification of information flow is not expected to lead to tractable analysis of real world software code.

Policy Specification Based on JML

A number of specification languages [25, 59, 28, 52] for *Java source code* build on the Java Modeling Language (JML [56]).

In [28], two forms of policies for explicit information flow are considered – positive flows and negative flows. An example of the former (resp. latter) is: the data in a formal parameter of a method in a class should (resp. should not) come from the return value of another method in another class. Implicit flows are not considered, and enforcing a flow policy amounts to guaranteeing a safety property. One way of specifying a policy is by using *sets*. For the example of the positive policy, one can specify: it is ensured (as a postcondition) that the return value must be in a set, and it is required (as a precondition) that the parameter should be from the same set. An alternative way is to use ghost fields of classes. For the same example, one can specify: it is ensured (as a postcondition) that a ghost field is set for the return value,

and it is required that the ghost field is set when the parameter receives its value. No implementation efforts are mentioned in [28].

In [25], the self-composition approach [11] is followed. A program *cmd* with public variable x is (termination-insensitively) secure if we have

$$\{x_1 = x_2\}\, cmd_1; cmd_2\, \{x_1 = x_2\},$$

where cmd_1 and cmd_2 are obtained from *cmd* by renaming the variables differently, including renaming x to x_1 in the former and to x_2 in the latter. This allows to capture noninterference using classical Hoare logic, with preconditions and postconditions specified in an extension of JML. The authors adapt the Krakatoa tool to process Java programs annotated in the extended JML, and generate proof obligations in the Coq proof assistant [55].

In [59], a different approach is proposed to specify noninterference properties in JML. This is via the keyword \old and *specification patterns*. For instance, the postcondition ensures low == \old(0) for a method states that the final value of the public variable low must be equal to the value of the constant 0 in the beginning, which reflects that the final value of low is not affected at all by confidential variables, guaranteeing termination-insensitive noninterference. It is also briefly investigated how to use the diverges clause and ghost variables of JML to reveal that information is leaked via termination. The author went on to develop a method to enforce termination-sensitive noninterference for Java via a *relational Hoare Logic*. A relational predicate can assert something about the value of a variable in two different states. The relational Hoare logic developed guarantees that the equality of the values of low memory locations is preserved by the big-step execution of methods. The standard Hoare triples are extended to Hoare n-tuples to capture different termination modes (e.g., diverging, terminating normally, with exception, etc.). These relational n-tuples can in turn be specified in JML for relations (JMLrel), and proved by a theorem prover such as PVS [47]. Note that proof automation is partial, and part of the proofs need to be accomplished via human interaction.

Among the above information-flow specification languages/methods based on JML, the one of [28] is closest in form to the direct specification of information flow policies via security domains, without a backend support for detecting implicit flows. In comparison, the ones in [25] and [59] are more indirect, considering either the sequential execution of two copies of a program or the simultaneous execution of the program with it self in two states.

In [52], both direct specification in terms of security domains and the detection of implicit flows are supported. This is by introducing a respects clause into JML* — an extension of JML. For each method, a set of views can be specified using the respects clause. Each view is a set of locations, with the intuition that locations outside of each view cannot interfere with locations in the view. A specification is then translated according to the self-composition approach, generating proof obligations in Dynamic Logic, that can be processed by the KeY tool (e.g., [1]). Support for sequential Java is claimed in [52]. It is unclear, as claimed in the same paper, how often the proof obligations can be automatically processed by Key.

Conditional Flow Policies

Some requirement languages provide the facility for explicitly stating *conditional flow policies*.

One line of work [15, 16, 14] enables the programmer to use "locks" in a program, and the information flow policy can be modulated by the state of locks. An example is: in an online auction with hidden bids, the bid of one bidder should not be revealed to other bidders before the auction closes. The status of the auction can be captured by the status of a lock variable. The policy of each bid is weakened to disclose the bid to everyone on opening the lock. Locks can be parameterized [16], which leads to the ability to model more complicated phenomena. An example is that a monadic lock can capture that a principal plays a particular role. Another example is that a dyadic lock can capture the trust relationship between two principals, leading to the ability to specify principal hierarchies, such as the ones in the Decentralized Label Model. The lock-based approach is developed to support Java, resulting in the Paragon language [14].

Another line of work [45, 38, 37] supports the dependency of flow policies on the current and future values of program variables, and the values of certain fields of messages communicated between concurrent processes. Thus the policies are often called content-dependent flow policies. Information is not allowed to leak via either the values of variables or the actual flow policies.

A different approach is to support conditional flow policies using explicit conditions and *implicit security domains* — via relational assertions [6, 10, 5]. If the value of a variable is required to be the same in a related pair of executions, then the variable is implicitly placed in the low/public security domain.

Presence Policies and Content Policies

Most often, security policies are specified for data content only: patient records, cryptographic keys, passwords, account balances, etc. Sometimes the presence of data, or meta-data, can also be subject to protection by information flow policies [43, 51, 22, 33, 48, 36, 13], since an attacker can be particularly concerned with whether one type of communication happens or not, whether a field exists in a table or an object, or how big an array or a table is. Most of existing work considering such tiered policies is done for theoretical languages.

RIFL

RIFL [26] is a specification language for information flow security with the distinguishing aim of achieving inter-operability of different tools for information flow analysis, different levels of abstraction, and different security semantics. The RIFL language is semi-formal, with rigorously defined syntax and intuitive semantics. A specification of security policies in RIFL comes in the XML format, and consists of an interface specification articulating the information sources and sinks of concern, a set of security domains (e.g., [41]) capturing the levels of protection, an assignment of categories of sources and sinks to domains, and a flow relation describing what the allowed flows between domains are. It is worth pointing out that the RIFL flow relations support intransitive information flow policies (e.g., [50]). RIFL consists language-independent modules and language-specific ones. The former can be used for high-level system models, while the latter for specific programming languages and intermediate languages. Language-specific modules exist for Java source code and Dalvik bytecode. RIFL has been supported by multiple tools including Joana [27], KeY [1], SuSi [49], and the RSCP security analyzer [40].

1.3 Analysis of Existing Specification Languages

In the CRISP project, the developed techniques need to be usable by software developers and architects. The less time engineers need to invest in learning a domain-specific formalism, the better chance that the corresponding technique can get exercised in practice. Thus specification languages based on logical formulas are unsuitable since the formulas are complicated at times and it is often difficult to understand them without understanding the underlying logic. In comparison, specification languages based on security levels/labels/domains are more “user-friendly”. The latter class of specification languages still permits sufficient generality and flexibility in that the security labels or domains can be tailored to specific application scenarios. Even if the labels have structures as “complicated” as those of the Decentralized Label Model, it is still easier for an average developer to use them than a specification language based on Dynamic Logic. Thus security labels/domains are an important language concept for the specification language to be developed in CRISP.

A related issue is automation. The additional cost of the software developers and architects in terms of time and energy in introducing security guarantees need to be minimized. To this end, policy specifications amenable to automated analysis and verification are desirable. The specification languages based on logical formulas usually permit high precision in the analysis in that false positives occur in fairly rare cases. However, such languages are supported by theorem-proving techniques, and precision comes at the price of the loss of full automation. A desirable approach is to rely on automated analyses and resort to theorem-proving for non-security properties only when and where needed [35]. In this case

the restricted amount of non-automated work proves safety properties that does not have to do with the specification language. The need for automation speaks again against the logically based languages, while in favor of the ones based on security labels and security domains.

A different issue is the granularity of specification. As required in the CRISP project, the approach to be developed should support Java source code. The specification language mainly addresses the level of applications rather than the operating system. Thus the security labels and security domains should apply to the entities in Java source code such as variables, fields, parameters, return values, classes, etc. It is desirable, although not required, for the specification language to be easily adapted to support other programming languages, as RIFL can [26].

The need for explicitly supporting declassification is not immediate. First of all, declassification can be indirectly expressed by intransitive information flow policies (e.g., [50]) with the help of additional security domains capturing down-graders. Second of all, once the flows between the security domains are clear, what needs to be done is to ascertain that the "dangerous" flows are required by the functionalities and in a sense unavoidable. This is not very different from what needs to be done when downgrading operations are explicitly used. Hence the support for declassification in certain existing specification languages (e.g. [43, 24]) does not necessitate the same requirement for the specification language to be developed for CRISP.

The need for explicitly support of presence policies and content policies is not immediate. From the specification viewpoint, a language supporting the use of security labels and security domains can readily encode the separation of presence policies and content policies. Hence the direct support for presence policies and content policies in certain existing specification languages (e.g., [51, 22]) does not necessitate the same requirement for the specification language to be developed for CRISP.

The need for specifying conditional flow policies (as in [16, 5]) is also not immediate. Compared with their non-conditional counterparts, conditional flow policies usually have more complicated specifications, and it is more difficult to articulate their security guarantees. The ability to specify conditional flow policies should rather be seen as one potential extension of the specification language to be developed, rather a requirement for it.

1.4 Requirements for the Specification Language

Based on the analysis of the existing specification languages, and the context of the CRISP project, we obtain the following requirements for our specification language to be designed later in the project.

Machine Readability.

The specification language needs to be machine-readable. This requirement is in fact shared by all the existing specification languages for data and information flow. In our context, the output of a tool for information flow analysis of some other component, or the assumptions made about the environment, needs to be fed to the tool performing the information flow analysis for the current application/component. Machine-readability is necessary for gluing together different tools, and/or different analysis to provide end-to-end, global results.

Human Readability.

It is an obvious requirement that the specification language needs to be easily readable and understandable by software architects and engineers. As argued in Section 1.3, this induces the requirement that the specification language should not be based on logical formulas, but rather on flows between security labels or security domains.

Tool Independency.

It is likely that the different software components in a large software system are analyzed by different tools. Thus it is desirable for the specification language to be a common language spoken by the multiple

analysis tools in use. It can be seen that tool independency is not a feature commonly possessed by the existing specification languages in the literature (RIFL [26] is an exception in this respect), although it is desirable in the context of CRISP.

Semi-formal.

The specification language is supposed to be semi-formal — with formally defined syntax, and informally described semantics. Formal syntax is necessary for machine readability. Informal semantics leaves room for different formal interpretations of the security requirements, such as with different noninterference properties. The accommodation of different formal semantics also caters for the diversity of tools that enforce different formal security properties.

XML Compatibility.

The use of the XML format eases the exchange of specifications between different analysis tools in use, and potentially between the multiple collaborators involved in the same project. It also facilitates the deep syntactical validation of the well-formedness of specifications using techniques like DTD (Document Type Definition) or XML Schema.

Java-targetedness.

As specified in the project proposal, Java is the programming language that should be targeted by the specification and analysis techniques for data and information flow in the project. Support of Java source code by the specification language to be developed is required.

Suitability for Interfacing with Library Analysis.

Apart from addressing the data and information flow within a software application, another important goal to be accomplished in the CRISP project is to combine the analysis of software components with the analysis of the use of resources such as the network, the file system, the clipboard of the underlying operation system, etc. Thus it is important for the application-level analysis developed by MAIS to interface well with the library analysis developed by the Software Technology Group at TU-Darmstadt. This includes the support of the analyses developed by both MAIS and STG at the level of the specification language.

Resource-awareness.

As mentioned above, the data and information flow policies of concern need to go beyond user applications and also cover software libraries that deal with system resources. Thus the specification language needs to be equipped with features that are concerned with resources. In more detail, out of the combined analysis by MAIS and STG, a system resource needs to be a viable source or sink of a flow. Resource-awareness makes the specification language to be developed not only applicable to application code, but also to system-level code — a rare feature in existing literature on specification languages for data and information flow.

Consistency of Security Classification According to Assumption.

As already stated in the introduction, the specification language needs to be able to convey requirements of data and information flow as well as the assumed flow of information in the environment of an application or one of its components. For the requirements to be sensible, the security classification (in the form of security labels or security domains) of sources and sinks of the application/component needs to be consistent with the assumption about the environment of the application/component, as specified in the assumption/output sub-language. For instance, if there is assumed information flow in the environment from a confidential sink to a source, then the source must be confidential as well in order for the information from the sink not to be leaked.

Common Notion of Sources/Sinks in Output Language and Requirement Language.
The data and information flow of a component in the environment can be used to check whether a requirement specification is sensible, as mentioned above. A different scenario is: in case such output has been produced from a component, it can be used instead of the component itself, for checking whether data and information flow requirements are satisfied by the component. To facilitate checking output against requirement specification in either case, it is desirable for the two to have a common notion of information sources and sinks.

Support for Identification of Common Interface of Two Components.
The information flow specification that is output from a component B is relevant for a different component A in that the "slice" of the output with respect to the *common interface* between the two components can be used, and suffices, as the assumption for the environment of A, when performing an information flow analysis of A. When considering an abstract environment for A, rather than a concrete component B, "slicing" the assumption about the environment over the common interface between A and the environment is sensible as well. Thus it is desirable for the specification language to facilitate the identification of the common interface of two components, and potentially be able to express such common interface if it is beneficial to do so.

1.5 Summary of Requirements for the Specification Language

We have identified a number of requirements for the specification language for data and information flow, to be defined at the next stage of the CRISP project. This is based on the setting and purpose for such a language given in the proposal of the project, the potential usage of such a language that can be derived from this setting, and an extensive literature survey on existing specification languages for specifying the flow of data and information in software systems.

Bibliography

[1] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. "The KeY Platform for Verification and Analysis of Java Programs". In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Revised Selected Papers*. 2014, pp. 55–71.

[2] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. "A verified information-flow architecture". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*. 2014, pp. 165–178.

[3] A. A. de Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. "Micro-Policies: Formally Verified, Tag-Based Security Monitors". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 813–830.

[4] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. "A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow". In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012*. 2012, pp. 369–389.

[5] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. "A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow". In: *Principles of Security and Trust (POST)*. 2012, pp. 369–389.

[6] T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve. "Specification and Checking of Software Contracts for Conditional Information Flow". In: *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*. 2008, pp. 229–245.

[7] G. R. Andrews and R. P. Reitman. "An Axiomatic Approach to Information Flow in Programs". In: *ACM Trans. Program. Lang. Syst.* 2.1 (1980), pp. 56–76.

[8] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. "Faceted execution of policy-agnostic programs". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013*. 2013, pp. 15–26.

[9] M. Balliu, M. Dam, and G. L. Guernic. "Epistemic temporal logic for information flow security". In: *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011*. 2011, p. 6.

[10] A. Banerjee, D. A. Naumann, and S. Rosenberg. "Expressive Declassification Policies and Modular Static Enforcement". In: *2008 IEEE Symposium on Security and Privacy (S&P 2008)*. 2008, pp. 339–353.

[11] G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[12] N. Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*. 2004, pp. 14–25.

[13] I. Bolosteanu and D. Garg. "Asymmetric Secure Multi-execution with Declassification". In: *Principles of Security and Trust - 5th International Conference, POST 2016*. 2016, pp. 24–45.

[14] N. Broberg, B. van Delft, and D. Sands. "Paragon for Practical Programming with Information-Flow Control". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013*. 2013, pp. 217–232.

[15] N. Broberg and D. Sands. “Flow Locks: Towards a Core Calculus for Dynamic Flow Policies”. In: *ESOP 2006, Held as Part of ETAPS 2006*. 2006, pp. 180–196.

[16] N. Broberg and D. Sands. “Paralocks: role-based information flow control and beyond”. In: *37’th POPL*. ACM, 2010, pp. 431–444.

[17] W. Cheng, D. R. K. Ports, D. A. Schultz, V. Popic, A. Blankstein, J. A. Cowling, D. Curtis, L. Shrira, and B. Liskov. “Abstractions for Usable Information Flow Control in Aeolus”. In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 2012, pp. 139–151.

[18] S. Chong and A. C. Myers. “Decentralized Robustness”. In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*. 2006, pp. 242–256.

[19] M. R. Clarkson, S. Chong, and A. C. Myers. “Civitas: Toward a Secure Voting System”. In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. 2008, pp. 354–368.

[20] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. “Temporal Logics for Hyperproperties”. In: *Principles of Security and Trust - Third International Conference, POST 2014*. 2014, pp. 265–284.

[21] Á. Darvas, R. Hähnle, and D. Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *Security in Pervasive Computing, Second International Conference, SPC 2005*. 2005, pp. 193–209.

[22] Z. Deng and G. Smith. “Lenient Array Operations for Practical Secure Information Flow”. In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*. 2004, p. 115.

[23] D. E. Denning and P. J. Denning. “Certification of Programs for Secure Information Flow”. In: *Commun. ACM* 20.7 (1977), pp. 504–513.

[24] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. “Model Checking Information Flow in Reactive Systems”. In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012*. 2012, pp. 169–185.

[25] G. Dufay, A. P. Felty, and S. Matwin. “Privacy-Sensitive Information Flow with JML”. In: *Automated Deduction - CADE-20, 20th International Conference on Automated*. 2005, pp. 116–130.

[26] S. Ereth, H. Mantel, and M. Perner. *Towards a Common Specification Language for Information-Flow Security in RS3 and Beyond: RIFL 1.0 - The Language*. Tech. rep. TU Darmstadt, 2014.

[27] J. Graf, M. Hecker, and M. Mohr. “Using JOANA for Information Flow Control in Java Programs - A Practical Guide”. In: *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar - 1. März 2013 in Aachen*. 2013, pp. 123–138.

[28] C. Haack, E. Poll, and A. Schubert. “Explicit Information Flow Properties in JML”. In: *Proc. WISSEC, 2008*. 2008.

[29] J. Y. Halpern, R. Fagin, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[30] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[31] B. Hicks, K. Ahmadizadeh, and P. McDaniel. “Understanding Practical Application Development in Security-typed Languages”. In: *22nd Annual Computer Security Applications Conference (ACSAC)*. Miami, Fl, 2006.

[32] *International Competition on Software Verification*. https://sv-comp.sosy-lab.org/2016/.

[33] N. Kobayashi. “Type-based information flow analysis for the pi-calculus”. In: *Acta Inf.* 42.4-5 (2005), pp. 291–347.

[34] M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. “Information flow control for standard OS abstractions”. In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007*. 2007, pp. 321–334.

[35] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. “A Hybrid Approach for Proving Noninterference of Java Programs”. In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015*. 2015, pp. 305–319.

[36] X. Li, F. Nielson, and H. R. Nielson. “Factorization of Behavioral Integrity”. In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security Part II*. 2015, pp. 500–519.

[37] X. Li, F. Nielson, and H. R. Nielson. “Future-dependent Flow Policies with Prophetic Variables”. In: *PLAS ’16*. 2016.

[38] X. Li, F. Nielson, H. R. Nielson, and X. Feng. “Disjunctive Information Flow for Communicating Processes”. In: *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Revised Selected Papers*. 2015, pp. 95–111.

[39] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. “Fabric: a platform for secure distributed computation and storage”. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009*. 2009, pp. 321–334.

[40] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. “Cassandra: Towards a Certifying App Store for Android”. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014*. 2014, pp. 93–104.

[41] H. Mantel. “A Uniform Framework for the Formal Specification and Verification of Information Flow Security”. PhD thesis. Saarbrücken, Germany: Universität des Saarlandes, 2003.

[42] H. Mantel. “Possibilistic Definitions of Security - An Assembly Kit”. In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop CSFW ’00*. 2000, pp. 185–199.

[43] A. C. Myers. “JFlow: Practical Mostly-Static Information Flow Control”. In: *POPL ’99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1999, pp. 228–241.

[44] A. Nanevski, A. Banerjee, and D. Garg. “Verification of Information Flow and Access Control Policies with Dependent Types”. In: *32nd IEEE Symposium on Security and Privacy, S&P 2011*. 2011, pp. 165–179.

[45] H. R. Nielson, F. Nielson, and X. Li. “Hoare Logic for Disjunctive Information Flow”. In: *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*. 2015, pp. 47–65.

[46] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[47] *PVS Specification and Verification System*. http://pvs.csl.sri.com/. Accessed: 2016-12-04.

[48] W. Rafnsson and A. Sabelfeld. “Compositional Information-Flow Security for Interactive Systems”. In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014*. 2014, pp. 277–292.

[49] S. Rasthofer, S. Arzt, and E. Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks”. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014*. 2014.

[50] A. W. Roscoe and M. H. Goldsmith. “What Is Intransitive Noninterference?” In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. 1999, pp. 228–238.

[51] A. Sabelfeld and H. Mantel. “Static Confidentiality Enforcement for Distributed Programs”. In: *Proceedings of the 9th International Static Analysis Symposium (SAS)*. LNCS 2477. Madrid, Spain: Springer, 2002, pp. 376–394.

[52] C. Scheben and P. H. Schmitt. “Verification of Information Flow Properties of Java Programs without Approximations”. In: *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Revised Selected Papers*. 2011, pp. 232–249.

[53] *SPARK 2014, expanding the boundaries of safe and secure programming*. http://www.spark-2014.org. Accessed: 2017-01-20.

[54] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. “Disjunction Category Labels”. In: *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems*. 2011, pp. 223–239.

[55] *The Coq Proof Assistant*. Webpage: http://coq.inria.fr.

[56] *The Java Modeling Language*. http://www.eecs.ucf.edu/ leavens/JML//index.shtml. Accessed: 2016-12-04.

[57] S. Tse and S. Zdancewic. “Run-time principals in information-flow type systems”. In: *ACM Trans. Program. Lang. Syst.* 30.1 (2007).

[58] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. N. Krohn, C. Frey, D. Ziegler, M. F. Kaashoek, R. Morris, and D. Mazières. “Labels and event processes in the Asbestos operating system”. In: *ACM Trans. Comput. Syst.* 25.4 (2007).

[59] M. Warnier. “Language Based Security for Java and JML”. PhD thesis. Nijmegen, The Netherlands: Radboud University, 2006.

[60] J. Yang, K. Yessenov, and A. Solar-Lezama. “A Language for Automatically Enforcing Privacy Policies”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’12. 2012.

[61] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. “Making information flow explicit in HiStar”. In: *Commun. ACM* 54.11 (2011), pp. 93–101.

[62] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. “Securing Distributed Systems with Information Flow Control”. In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008*. 2008, pp. 293–308.

[63] L. Zheng and A. C. Myers. “End-to-End Availability Policies and Noninterference”. In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*. 2005, pp. 272–286.

2 The Java Native Interface - A Literature Review of Analytical and Constructive Research

In CRISP, we develop new methods to analyze the capability usage of libraries implemented in C and C++ which are closer to the system than libraries written in Java or scripting languages. As languages like Java provide abstractions over the details of the operating system layer and guarantees on the execution of programs written in these languages (e.g., memory safety), they are considered *safe* languages. However, these abstractions have to be bound to system functionality to allow programs to have effects on the system (e.g., writing a file). The authority and ability to perform such an effect is called a capability.

Also, as values have to be passed to system functions, low-level attacks can also use programs written in those *safe* languages as their entry point into a system. This makes these programs as vulnerable as programs written in languages without safety guarantees.

One particularly interesting case is the Java platform with its *Java Native Interface (JNI)* to bind libraries and operating system functionality written in C or C++. We chronologically outline the complete course of research in the area of JNI security and safety in the following.

2006

While there has been preceding work about other FFIs, Michael Furr and Jeffrey S. Foster were among the first to specifically deal with the JNI's security. In 2006 they published their paper "Polymorphic Type Inference for the JNI" [65] in which they present a type-inference system that is used to check the type safety of programs that use the JNI. As class names, field names and method signatures are represented by strings in the JNI, they track the flow of string constants through the program. This allows them to assign type signatures to the JNI functions and to infer them for most user-defined functions. The constraints they use for inference are equations, unification constraints are assuring e.g. that for `type variableName = function(argument)`, `type` matches `function`'s return type. Functions can even be treated polymorphically using instantiation constraints. These instantiation constraints are more complex equations that model substitutions for quantified variables for each instantiation site. They solve these constraints with a variant of Fähndrich et al.'s worklist algorithm for semi-unification. They present an implementation of their concept, their type inference tool works on the C code and issues warnings upon encountering a type error. A particularity lies within the fact that, while e.g. `jstring` is just an alias for `jobject`, they enforce the use of the correct alias, i.e. map `jstring` to `String` and not `Object`. Their implementation further ignores string operations that could hide type information, potentially introducing imprecisions, yet they claim that this is not common practice anyway. They run their analysis on a set of programs using the JNI, discovering 155 erros which stem from declaring a C function with an incorrect arity, software rewrites, wrong parameter type specifications, type mismatches and incorrect namings. They produce 44 false positives and in 22 cases the analysis had insufficient information about a string.

In the same year, Gang Tan et al. published their paper "Safe Java Native Interface" [80], presenting their SafeJNI system which, by static and dynamic checks, retro-fits native C methods to enhance their safety and captures safety invariants whose satisfaction aims at guaranteeing a safe inter-operation between Java and native code. More precisely, they leverage CCured, a system that statically and dynamically checks C code, to provide enhanced functionality for checking C code that uses the JNI. CCured already classifies pointers as SAFE (no pointer arithmetic, not type cast), SEQ (pointer arithmetic, not type cast, gets bound checked) and WILD (dangerous type casts, e.g. integer to pointer, information about original type attached). The authors add two more, HNDL (cannot be read or written, pointer only for passing to JNI calls) and RO (read only, prevents tampering with the JNI interface pointer). Moreover, they insert dynamic checks into the code, ensuring that objects passed from C to Java are of the right class, that Java methods are invoked with the correct number and the correct types of argu-

ments, that Java's access control is enforced (`private`, `public`, etc.), that Java array accesses are bounds checked and that exceptions are correctly handled by the native code. In addition, they present two concepts to ensure safe memory management, one of them involving a garbage collector, that keeps track of the JNI dynamic memory functions and ensure correct usage. The authors evaluated their system by injecting unsafe code into "otherwise safe programs" and showed that all vulnerabilities were caught. To evaluate performance, they applied their system to the Zlib library and showed that execution time is increased by 63%. This is more than the 46% an unmodified CCured incurs but less than the 74% they measured for a Java implementation of the library. Finally, they discovered a vulnerability in the library relating to a function receiving an integer as an argument that is then cast to a pointer and formally proofed that with their system native code will not violate memory safety or access Java's memory arbitrarily.

2007

In 2007, Gang Tan and Greg Morrisett proposed "ILEA: Inter-Language Analysis across Java and C" [82], a framework that models C code on an abstract level to help Java analyses understand its behavior and make inter-language analysis more precise. They introduce an extension of the Java Virtual Machine Language to model C code and present an automatic specification extractor for C code. Namely, they add three more operations, `choose` τ which chooses a random object of type τ, `mutate` which mutates an existing object and `top` which may have any type-preserving effect on the JVM. Figure 2.1 illustrates this.

```
for (int i = 0; i < choose(int); i++) {
        Object x = choose(Object);
        mutate(x);
}
```

Figure 2.1: An example of ILEA-modeled C code

The code mutates randomly many existing objects but can e.g. not allocate memory. Their automatic specification extraction is based on the CIL framework, a LLVM Bitcode-like intermediate representation, to get rid of many of C's complexities. The extractor can deal with most of C's functionality, in doubt `top` gives the most conservative approximation of behavior. They evaluate on four programs by modifying the Jlint Java analysis to understand C code and search for null-related bugs. They find 25 of them in the programs with one false positive.

Also in 2007, Martin Hinzel and Robert Grimm presented "Jeannie: Granting Java Native Interface Developers Their Wishes" [67]. Jeannie is a new language that nests Java and C code within each other to enhance productivity, safety, portability and efficiency. The combination of the two languages resides in the same file and compiles down to JNI code. Code follows mostly the two languages' syntax, a backtick operator indicates switches between the two. Jeannie does not need fully qualified names for Java objects and the compiler can verify type and name consistency across languages boundaries. `throws` clauses are checked to declare all exceptions, even in native code, resources are managed automatically like in Java. The compiler chain the authors provide takes a Jeannie file with mixed code and produces, after preprocessing, syntax checking etc. a shared object file and Java bytecode. Jeannie avoids manual synchronization with `MonitorEnter()` and `MonitorExit()` by allowing `synchronized` statements in C code. Even exceptions can be handled Java-style in C code through `try` statements. Two special functions for copying regions of arrays and strings between the two languages with adequate encoding speed up bulk access. A with-like statement allows disciplined access to Java arrays and strings, the error-prone `Get.../Release...` pairs can be avoided. The evaluation, in which they ported a JNI program to Jeannie, shows that the amount of code in JNI programs can be significantly reduced. The performance penalty compared to a common JNI implementation ranges from 4% to 86% in their test cases, some

examples even showed a 5% speedup. Especially array accesses through the with-like statement showed to be up to 75 times faster.

2008

In the following year, Gang Tan and Jason Croft published ”An Empirical Security Study of the Native Code in the JDK” [81] where they describe how they established a taxonomy of bugs in the Java Development Kit’s native code to perform a systematic search to discover bugs and finally propose remedies for them. The authors discovered 59 previously unknown bugs, leveraging existing tools and providing new ones. To scan for common C bug patterns they use Splint, ITS4 and Flawfinder (static analysis tools that highlight probable bug patterns) and for JNI specific bugs they provide grep-based scripts and scanners implemented in CIL. They manually inspected the results as all of the analyses provided high false positive rates (97.5-99.8% for the off-the-shelf tools). They focused on the `share/native/java` and `solaris/native/java` folders. Their heavy reliance on manual inspection made this limitation necessary. They stress that the high rate of false positives is also due to the fact that a single language analysis is less precise than an inter-language analysis would be, as the former could e.g. not know if inputs that are unchecked in C code might get a sufficient sanitization in the Java part and thus has to conservatively assume potential threat.

Yuji Chiba introduced ”Java heap protection for debugging native methods” [64] the same year, a feature that detects and locates invalid memory references to the JVM heap. As these can stem from either the JVM implementation or the native code, the latter being hard to trace, it is supposed to help developers find bugs. The feature uses page protection which prevents threads executing native methods from referring to the JVM heap directly, as this should only happen through the JNI. This is mandated by most common JVMs like Hotspot (used in the OpenJDK) which this paper uses. As opposed to mainstream operating systems, the protection feature controls permissions by thread and not by process. The author stresses that the feature cannot universally detect bugs but only those specific to heap reference errors. It further requires modification of the OS (about 500 LOC) and the JVM (about 60 LOC) which limits practicality. The performance overhead introduced depends heavily on the frequency of native method calls, ranging from 2% for sporadic use to 54% for frequent use in the author’s experiments.

Also in 2008, Goh Kondoh and Tamiya Onodera released their paper ”Finding Bugs in Java Native Interface Programs” [68]. They present static analysis techniques to find bugs in programs using the JNI, focusing on four distinct error patterns: missing statements for explicit JNI exception handling, mistakes in resource management due to erroneous manual allocation, local references that are not converted to global references when living across multiple method invocations and calls to disallowed JNI functions in critical regions. All four patterns are caught by typestate analysis, automata-like state transition systems that ensure the correct execution order of operations on an instance of a given type. They implement their analyses with BEAM (a Lint-like analysis tool that leverages theorem proving to check the feasibility of potential errors) to check four open-source programs that use the JNI (Harmony, Java Gnome, Gnu Classpath and Mozilla Firefox) for errors. They discovered 86 JNI-specific bugs (and 114 non-JNI bugs).

2009

In 2009, Siliang Li and Gang Tan published ”Finding Bugs in Exceptional Situations of JNI Programs” [74], proposing a static analysis framework to examine exceptions in JNI programs and report if errors occur with respect to incorrect exception handling (performing cleanup work, explicit returning etc.). Their framework consists of three main components. The first is a standard inter-procedural dataflow analysis algorithm with extra conditions to track exception states in which JNI exceptions are pending. The second component checks the succeeding operation for all these states against a whitelist of allowed operations and uses a taint analysis to simulate the propagation of faults. The third component transforms the input program after a warning has been issued and then recomputes the dataflow equations to avoid duplicate warnings for the same error. As their scheme of detecting erroneous exception handling

resembles Kondoh and Onera's from the previous paragraph, Li and Gang justify why their work is better. They outline that the previous approach was incomplete as it would only catch exceptions thrown in a particular way (`Call<Type>Method`) while the JNI allows for many more possibilities (`Throw/ThrowNew`, `Get<Type>ArrayElements` etc.). They claim that their new approach is more complete. The code they evaluate on is the one checked in the empirical study from 2008 plus the `share/native/sun` folder and three non-JDK programs (java-posix, java-gnome and java-opengl). Thanks to their third component, they have low false positive rates of a 15.4% average over all the programs and discover 716 errors.

Also in 2009, Byeongcheol Lee et al. released "Debug All Your Code: Portable Mixed-Environment Debugging" [69] which is a composition approach for simple, portable and powerful mixed-environment debuggers. An intermediate agent interposes on language transitions and controls and reuses single-environment debuggers. The authors exemplify this by introducing Blink, a debugger for Java, C and Jeannie. Blink requires 9000 lines of new code but greatly facilitates debugging JNI applications. The debugger features execution control (the ability to start, stop, halt at breakpoints and single-step through a program), context management (keeping track of source code and call stack traces) and data inspection (allowing the inspection of variables and values at certain points in the program execution). There are soft-mode debuggers in which the debuggee has to process certain basic commands for the debugger and hard-mode debuggers where this is not the case. Most C debuggers are hard-mode, Java debuggers typically soft-mode. The latter, influencing the debuggee's behavior, can lead to undesired side-effects. Blink is soft-mode, but warns the user of inconsistencies that may arise. Blink has advanced features like the environmental transition checker which detects uncaught exceptions and unexpected null values and read-eval-print loops which allow interpretation of arbitrary source language expressions based on the current application state. The authors' evaluation of Blink showed their debugger to be portable to different OSs and JVMs and to have a low time overhead of an average of about 10% for their test programs.

2010

Joseph Siefers, Gang Tan and Greg Morrisett presented "Robusta: Taming the Native Beast of the JVM" [76] in 2010, introducing a novel framework to provide safety and security for Java applications and applets through software-based fault isolation which places the native code into a sandbox. Their sandboxing is based on Google's Native Client (sandboxing technology that allows execution of untrusted C/C++ code in a browser), extending it with support for dynamic linking and loading of native libraries. To make the code work inside the sandbox, fake interface pointers which reside in an unmodifiable memory region, called "trampolines", redirect JNI calls to trusted wrappers outside of the sandbox which perform safety and security checks. For system-calls inside the sandbox, system-call trampolines redirect to system-call wrappers that connect to the JVM's security manager to check permissions and leverage Java's existing security infrastructure. Trampolines are one-way and the only way through which control can escape the sandbox, entry into the sandbox is provided by so called "springboards" without further functionality. They experimentally evaluated their results on five different JNI programs (`java.util.zip`, libec, `java.lang.StrictMath`, libharu and libjpeg) to see that performance overheads are one-figure for applications with few context switches while they surge for applications with many to over 700%.

The same year, Byeongcheol Lee et al. published "Jinn: Synthesizing a Dynamic Bug Detector for Foreign Language Interfaces" [70]. They show how dynamic analyses can be synthesized from state machines to detect FFI violations and demonstrated this on the example of the JNI. They identified three big classes of rules that enforce safety for the JNI: JVM state constraints, type constraints and resource constraints that each can be represented by a type of state machine. They use state machines to synthesize dynamic analyses from them, the synthesizer outputs Jinn, a shared object file automatically loaded by the JVM through the JVM tools interface which then monitors program events and state, throwing an exception upon encountering misbehavior. The three constraint classes are subdivided to form nine different automata specifications that model the following checks: check if the current

thread matches the used JNI pointer's thread, check that no exceptions are pending and no critical region is open upon sensitive calls, check parameters that may not be NULL to be not NULL, check that parameters match signatures, check that no final fields are written, check that no overflows, leaks, dangling references or double frees occur. Their evaluation showed an average of 14% runtime overhead for diverse benchmarks. They ran Jinn over three widely-used programs and uncovered two overflows of local references and one dangling reference in Subversion, one nullness-bug and one dangling local reference in Java-gnome and one violation of a typing constraint in Eclipse.

2011

In 2011, Siliang Li and Gang Tan introduce "JET: Exception Checking in the Java Native Interface" [75], a static analysis framework that extends the exception checking of Java (every method that can throw exceptions must have an appropriate `throws` clause) to native code. Their two-stage process first automatically eliminates all non-interface code from the code to be analyzed as only interface code uses JNI functions to inspect and change Java state. This speeds up analysis by a factor of about 60 for their test programs. Their following fine grained analysis tracks exception states to check for erroneous or lacking exception throwing declarations. It is path-sensitive to account for the correlation between exception states and other execution states, e.g. JNI methods throw exceptions *and* return special values upon encountering failures, if code checks one of them, subsequent state has to take this correlation into account. The analysis is further context-sensitive as operations using the JNI often involve multiple calls. They ran their analysis on 15 different JNI programs and the tool reported 18 warnings of which 12 were true bugs. The false positives were due to the inability of the program to track exception types stored in string variables so the conservative assumption of `java.lang.Exception` as the type resulted in a mismatch.

Also in 2011, Du Li and Witawas Srisa-an presented "Quarantine: A Framework to Mitigate Memory Errors in JNI Applications" [71], a runtime system that can identify objects accessible by native methods and move them to a special "quarantine" area of the heap. This allows the application of heap protection techniques for native languages that did not work for Java before. The separation protects against out-of-bounds copying, buffer overflows, mistakes by pointers as integers and memory leaks by erroneous dynamic memory management. The separate heap is much smaller and as many heap protection techniques work best for small heaps, Quarantine enables many of them. The authors do not present or mandate any specific protection mechanisms but keep their system configurable and open. Their implementation builds on the Jikes RVM, a Java Virtual Machine written in Java destined for research purposes, and features a special hybrid garbage collector as well as read- and write-barriers for separation. The runtime performance penalty Quarantine incurs showed to range from 4% to 24% for diverse tests, best performance could be reached with a heap five times the minimum required size.

2012

In 2012 multiple papers on JNI security were released, among them "JATO: Native Code Atomicity for Java" [72] by Siliang Li, Yu David Liu and Gang Tan in which they present JATO, a system that enforces atomicity of native methods when invoked by Java through the JNI. The addition of atomicity aimed at facilitating the design of parallel programming models using the JNI. To that extent, JATO identifies the set of Java objects that need to be protected for atomicity with a constraint-based inference algorithm which can extract memory-access constraints for both Java and C uniformly. This is based on the observation that despite different syntax and semantics both languages access memory in a similar manner. In their standard approach, atomic blocks are whole native methods. They introduce and evaluate different locking schemes, locking around just the regions that access critical objects proved most efficient in their prototype implementation. This implementation generates the Java-side constraints with Cypress (a static analysis framework with focus on memory access patterns) and native-side constraints are generated with the help of CIL. Even with their most optimized locking scheme, the locking introduced more overhead than performance was gained through parallelization for most test applications.

In addition, Gang Tan presented ”JNI Light: An Operational Model for the Core JNI” [79] the same year. It introduced JNI Light (JNIL) which is a formalization of a common model of (a part of) Java and its native side. The Java-side language is a subset of the JVM Language, the native-side language is a RISC-style assembly language augmented with JNI functions. JNIL models a shared heap, but as Java takes a high-level view on the heap as a map from labels to objects, while native code takes a low-level view as a map from addresses to primitive values, a compromise is needed. JNIL adopts a block model where a heap is a map from labels to blocks and a block is a map from addresses to primitive values. This way, a block may hold a Java object representation or a memory region allocated by native code, simplifying garbage collection while accommodating to the low-level view. JNIL does not commit to any object-representation strategy, a representation function maps objects to a block. Method calls from both languages share a call stack whose frames are either Java frames or native frames. Java exceptions are indicated by a special exception frame at the top of the method call stack while a native exception is recorded as a special configuration of a native frame. According to real behavior, JNIL unwinds the stack for Java exceptions and continues execution for JNI exceptions. JNIL records Java references available to a native method in a set associated with a native frame such that automatic freeing is possible. Tan presented formal semantics for his model, admitting to a number of simplifications like excluding arrays and assuming a specific calling convention. To show JNIL's utility, the author formalized a static checking system in JNIL which checks native code for violations of Java type safety and presented a soundness theorem.

Also in 2012, Mengtao Sun and Gang Tan published ”JVM-Portable Sandboxing of Java's Native Libraries” [77], introducing Arabica, an enhancement of their previous sandboxing approach Robusta to be portable between different Java VMs by avoiding modification of VM internals. Arabica relies on a combination of the Java Virtual Machine Tool Interface (JVMTI) and a layer of stub libraries. Just like the JNI, the JVMTI is supported by nearly all JVM implementations. A JVMTI agent registers callback functions for the native-method-bind events which occur when a native method is bound to the adress of a native library function that implements the native method. The callback then initializes the safety checks. Like Robusta, Arabica provides JNI interface function wrappers. Yet this is not enough for a comprehensive sandboxing as the JVMTI offers no support for a native-library-loading events, rendering them uninterceptable. Arabica thus introduces a level of indirection through trusted stub libraries which perform native library loading, symbol resolution, native method calling and returning. To evaluate, the authors sandboxed the standard native libraries in the JCL, reducing the JCL's trusted code base and making it possible to evaluate performance with standard Java benchmark suites. Instead of putting all code into the sandbox, they only sandbox native code that is not part of Java's security infrastructure for safe separation and portability. As Arabica also leverages Java's security manager for system calls from native code, they assign permissions to the native code on a per-package basis, choosing the minimum of required permissions. Evaluation on the same set of JNI programs they evaluated for Robusta, shows that the performance overhead for context switch-intensive applications is more than doubled compared to Robusta, rocketing it to around 1600% (with slight differences between different JVM implementations). Yet for common Java benchmarks, overheads of between 3% and 12% can be seen, in one case more than 110% are observed. This shows that the increased compatibility comes with additional costs to the already high costs of security through sandboxing.

2013

In 2013 ”JNICodejail - Native Code Isolation for Java Programs” [66] by Behnaz Hassanshahi and Roland H. C. Yap was released. They present a JNI sandboxing approach as an alternative to Arabica. Unlike Arabica, it does not use Software Fault Isolation but a tightly integrated, library-isolating memory model. It is built on top of CodeJail, an application-transparent isolation mechanism for untrusted libraries that need to interact closely with the main program. Trusted and untrusted code share an address space, if untrusted code wants to write to the trusted code's memory, a copy is created in the untrusted memory and changes have to be explicitly synchronized. This synchronization can be used for

safety checks, system calls are restricted by a policy. They map this concept to the JNI, considering native methods as untrusted library code and the Java part as trusted. A key aspect is that if an untrusted native method only reads and does not write trusted Java memory, direct access can be granted which improves performance while Arabica redirects any calls. This also enables JNI functions that return direct pointers outside the sandbox (untrusted) region to work with good performance. Evaluation was inconclusive, sometimes Arabica performed better, sometimes JNICodeJail.

2014

The list of this year's papers can naturally not claim completeness, yet we present what has been released so far. Siliang Li and Gang Tan published "Exception Analysis in the Java Native Interface" [73] in which they present a static analysis framework called TurboJet which unifies, improves and extends their 2009 concept for finding exception handling errors and their 2011 JET system for finding inconsistent exception declarations. They improve precision and efficiency of their former approaches, conduct a more thorough evaluation featuring more benchmarks and add an Eclipse plug-in which supports a software developer via a GUI in spotting the mistakes TurboJet detects. They compare TurboJet with their previous empirical security study and state that they find nearly thrice as many true bugs with a lower number of warnings, thus heavily decreasing the false positives.

Also this year, Mengtao Sun and Gang Tan introduced "NativeGuard: Protecting Android Applications from Third-Party Native Libraries" [78]. They observed that Android poorly manages the security risk Java applications' native libraries pose. Most Android applications are written in Java and they make pervasive use of native libraries. In a survey they found that 86% of the top 50 applications in the Google Play store (the main source of software for most Android users) use at least one native library. The increase in application complexity renders the need for native performance more pressing. The authors' NativeGuard system separates a Java application's native code into a second application, leveraging Android's process-based protection. This second application with native code gets minimal privileges, it does not share the full privileges of the Java part anymore. To separate the application's code, the application's launcher is modified, proxy native libraries which reside with the Java part and redirect to the native part in combination with inter-process communication (IPC) allow the two parts to communicate. Stub libraries in the native part are required to jump back to the Java part to e.g. resolve opaque references. Upon JNI function calls, runtime type checking is performed. Much of this resembles the Arabica approach, yet the Dalvik JVM lacks the JVM Tool Interface and thus a new approach is necessary. Their experiments showed a performance overhead ranging between 0.54% to 17.16% for their benchmarks, another experiment showed a strong correlation between performance overhead and context switches, like for Robusta / Arabica.

Bibliography

[64] Y. Chiba. “Java Heap Protection for Debugging Native Methods”. In: *Sci. Comput. Program.* 70.2-3 (Feb. 2008), pp. 149–167.

[65] M. Furr and J. S. Foster. “Polymorphic Type Inference for the JNI”. In: *Proceedings of the 15th European Conference on Programming Languages and Systems*. ESOP’06. Vienna, Austria: Springer-Verlag, 2006, pp. 309–324.

[66] B. Hassanshahi and R. H. C. Yap. “JNICodejail: Native Code Isolation for Java Programs”. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ ’13. Stuttgart, Germany: ACM, 2013, pp. 173–176.

[67] M. Hirzel and R. Grimm. “Jeannie: Granting Java Native Interface Developers Their Wishes”. In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 19–38.

[68] G. Kondoh and T. Onodera. “Finding Bugs in Java Native Interface Programs”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA ’08. Seattle, WA, USA: ACM, 2008, pp. 109–118.

[69] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. “Debug All Your Code: Portable Mixed-environment Debugging”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 207–226.

[70] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. “Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces”. In: *SIGPLAN Not.* 45.6 (June 2010), pp. 36–49.

[71] D. Li and W. Srisa-an. “Quarantine: A Framework to Mitigate Memory Errors in JNI Applications”. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ ’11. Kongens Lyngby, Denmark: ACM, 2011, pp. 1–10.

[72] S. Li, Y. Liu, and G. Tan. “JATO: Native Code Atomicity for Java”. English. In: *Programming Languages and Systems*. Ed. by R. Jhala and A. Igarashi. Vol. 7705. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 2–17.

[73] S. Li and G. Tan. *Exception Analysis in the Java Native Interface*. 2014.

[74] S. Li and G. Tan. “Finding Bugs in Exceptional Situations of JNI Programs”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS ’09. Chicago, Illinois, USA: ACM, 2009, pp. 442–452.

[75] S. Li and G. Tan. “JET: Exception Checking in the Java Native Interface”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 345–358.

[76] J. Siefers, G. Tan, and G. Morrisett. “Robusta: Taming the Native Beast of the JVM”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, Illinois, USA: ACM, 2010, pp. 201–211.

[77] M. Sun and G. Tan. “JVM-Portable Sandboxing of Java’s Native Libraries”. English. In: *Computer Security – ESORICS 2012*. Ed. by S. Foresti, M. Yung, and F. Martinelli. Vol. 7459. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 842–858.

[78] M. Sun and G. Tan. “NativeGuard: Protecting Android Applications from Third-party Native Libraries”. In: *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*. WiSec ’14. Oxford, United Kingdom: ACM, 2014, pp. 165–176.

[79] G. Tan. “JNI Light: An Operational Model for the Core JNI”. In: *Proceedings of the 8th Asian Conference on Programming Languages and Systems*. APLAS’10. Shanghai, China: Springer-Verlag, 2010, pp. 114–130.

[80] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. “Safe Java native interface”. In: *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*. 2006, pp. 97–106.

[81] G. Tan and J. Croft. “An Empirical Security Study of the Native Code in the JDK”. In: *Proceedings of the 17th Conference on Security Symposium*. SS’08. San Jose, CA: USENIX Association, 2008, pp. 365–377.

[82] G. Tan and G. Morrisett. “Ilea: Inter-language Analysis Across Java and C”. In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 39–56.