Localising Unsafe Software Resource Usage with Typed Code Model

Lavrentii Tsvetkov, Anton Spivak
ITMO University
Saint Petersburg, Russia
lavrentii.tsvetkov@corp.ifmo.ru, anton.spivak@gmail.com

Abstract—**The article presents a method for quality assurance on resource leakage by defect search automation using developed model of program code. Resources requested by the program are identified and assigned extended types storing state markers. Detection occurs by assignment of this markers to elements of code model according to resource management functions, evidencing resource availability. Further analysis is taken place by propagation of the markers according to the rules of a model. Rules are structured a in way that prevents unsafe use of resource handles. Inability to apply specific rule at analysis stage signifies that resource is used in unsafe way, creating potential security flaw in a program.**

I. Introduction

Correct resource management is essential part in software development. Despite wide spread of hight-level programming languages with automatic memory management, they still do not provide tools for ensuring correct handing of many other resources, such as file handles, network connections, and other objects with manual internal state management.

By violating expected order of resource management operations, software defect is introduced, opening a gap for vulnerability to affect a program. Consider unused memory block that was not deallocated by usual means. Access to this memory can still be done without alert of operating system, leading to information leakage. Accumulating unclosed network connections may result denial of service when open connection limit will be reached by operating system.

To prevent such events, strictness of resource management must be enforced or at least maximized. There are several ways to increase strictness of resource management.

First way is to restrict use of resource access primitives and provide higher-order access functions. In this scenario, basic functions such as $\mathbf{open}\,/\,\mathbf{close}$ are replaced with high-order combinators such as $\mathbf{withFile}$, automatically closing file after provided callback function is finished, or by $\mathbf{readFile}$, reading full file content at once [1]. Clearly, this approach can not be applied to wide variety of software by many reasons. In particular, full content stream can not be read in case of interactive message exchange, and opening many files with higher-order function will lead to stack depletion.

Second way is to include in a programming language new statements, assuring correct resource deallocation. Examples of this approach are C++ destructors [2] and deferred functions in Go [3]. This approach ensures correct handling of resource if corresponding statements are used. Drawback of this approach is that resource is bound to lexical scope it was declared in. From this follows impossibility to transfer resource handle copy upward in call stack without creating alias handles, introducing potential inconsistency in their use. To mitigate this, more entities are introduced in different languages, including move semantics, reference counting and null states.

Third approach is to introduce additional compilation stage validating correctness of resource usage using state markers associated with resource handle. Because markers are assigned to resource protocol primitives (such as POSIX functions), this method is not bound by particular programming language. This approach can validate programs statically, allowing not only future software to be checked, but also existing ones.

In this paper third approach is considered for localising unsafe resource usage. Proposed markers are assigned using “resource type” introduced to program language type system. Since programming languages of interest already have fairly solid type systems, to help them in detection of unsafe resource usage, extended types should be implemented, preserving special properties that need to be considered when attempt of modifying resource state is made. This is a typical use case for rich type system, that is system capable of tracking sophisticated interactions of variables, conditions, etc. at compile time.

Rich type systems are used greatly in functional languages, such as Haskell, where monads are used to limit computational side effects. Languages like Agda [4] and Idris [5] allow even more precise restrictions on operations to be defined using dependent types. On the other hand, type systems for imperative languages were not improving for last decade until Rust language was released [6]. Taking some inspirations in functional programming, Rust features immutable-by-default variables and function parameters. Moreover, key feature of Rust — “borrowing” — ensures exclusive write access to referenced resources. This mechanism augments main type system of Rust by effectively implementing an affine type system on variable scopes itself. This provides strict solution to resource aliasing problem.

Augmented type system approach is preferable due to limited concern of actual type system used by programming language. This allows to implement the core of defect localisation method regardless of details of particular programming language, opening the way for language-agnostic analysis.

Following issues are addressed in this paper:

- Kinds of entities that can be modeled with linear resource management operations: While main objects that approach is applicable for are file and network handles, many other objects that can benefit from extra information tracked at type level are candidates for using introduced resource type.

- Resource aliasing: Programs that address same resource by interleaving access from different aliases can not be analysed using simple type systems. Problem arises because usage of one of the aliases must affect types of other aliases when resource management functions are called. Since our approach is targeted at existing programs that generally could not be easily restructured, developed system must be capable of tracking aliased resources and properly handle them.
- Ownership transfer: Imperative nature of programming languages of interest simply does not allow to apply functional variable-as-proofs principle, meaning that availability of resource is not guaranteed by resource handle value not equaling special empty value. This means that in presence of correct aliasing tracking, ensuring resource deallocation must be performed only when all aliases are unreachable or already reset to empty values. This requirement is too strict, disallowing resource release in subsequent procedures that nest more than most upper-level resource alias. This is a common practise in programming to have paired resource operations to not match clearly on same procedure nesting level. By ownership transfer down the nesting level we minimise positive alerts in programs that are correct according to unstated assumption by programmer.
- Conditional resource management: Most part of dangerous resource usage come from conditional statements. Error made in rarely used condition branch can stay undetected for a long time. Vulnerabilities often rely on such code to initiate intrusion into a program. There are many reasons why conditional resource management is present in a program. One is optimisation attempt on resource (de)initialisation. Other is improper handing on error conditions, such as returning early from function with part of resource allocated.
- Analysis localisation: Implementation of type system with dependent types is faced with a fact of verifying all possible routes of conditional statements quickly leads to combinatoric explosion. This effect can not be fully eliminated but can be mitigated by limiting full type inference by boundaries of semantically grouped parts of a program — procedure declarations, representing its structure as black box with pre- and post- conditions on required state of resource. Not only meaningful to human result is produced, but also derived type can be reused without rescanning inner procedure structure. This is not a case for other structures that can be represented internally as functions, such as loops as dependence on previous stage can result although correct but obscure type expression.

First, in Section II existing works on localising software defects are analysed, discussing their pros and cons. In Section III outline how different kinds of objects can be rendered as resources by mapping their main operations to resource operations. Next, in Section IV type model of a resource is presented as well as rules that govern defect localisation method. In particular, in Section IV-A basic rules of resource management are presented that are sufficient for diagnosing linear resource management that does not involve conditionals. In Section IV-B more detailed rules are shown that allow smooth integration over different conditional statements by using dependent typing rules on corresponding resource states. Section IV-C describes how presented approach allow to limit depth of type search in case of nested procedure invocations by factoring out irrelevant type elements from procedure types. Section IV-D discusses how specific resource usage pattern — implicitly transferring ownership — can be inferred to minimise false-positives in defect localisation. Finally, in Section VI future directions of improvement are discussed.

II. Related works

Related works on localising unsafe resource usages can be categorised to dynamic and static approaches.

Valgrind [7] is dynamic binary analysis tool with primary use of locating erroneous memory operations such as out-of-bounds access, double-free, uninitialised use of memory and etc. It can be specified to report unclosed file descriptors on program exit. If such descriptors exist, stack trace reported to the user, identifying location where descriptor was opened. Unfortunately, there is no possibility to identify reason why descriptor was not closed properly.

Address Sanitizer [8] is another dynamic analysis tool. It is specialised to find errors corresponding to one type of resource — memory. It can detect out-of-bounds access, use-after-free errors, as well as report leaked memory at the end of a program. It maps available memory to special shadow zone, tracking current status of memory cell. When erroneous access is made, it is detected immediately and current stack trace is reported. Using compressed shadow state encoding, average program slowdown is 73%. Address Sanitizer has no false-positive alerts.

Both Address Sanitizer and Valgrind are very effective in locating unsafe memory usage patterns, but other types of resources are not tracked. Moreover, error could be detected only when appropriate code is executed, leaving error undetected if particular code section was left uncovered by performed tests. This is limitation of dynamic analysis, because there is no way to preform analysis on non-executed parts of code.

Static analysis methods do not suffer from this restriction. There are several systems that use dependent typing to ensure different properties on type level.

Hoare Type Theory [9] is implementation of Hoare Logic on type level. Imperative computations are executed in Hoare monad. Postconditions in Hoare triples can depend on returned value, allowing caller code to identify measures needed to cleanup allocated resources. HTT uses two-stage type inference to mitigate undecidability issue arising in some programs.

Ynot [10] is logical extension over HTT. Ynot is implements HTT rules in Coq proof assistant. Main goal is to allow imperative programs with side-effects to be implemented in purely functional languages. Dependent typing that is required for HTT is provided by Coq. With this approach authors have implemented several imperative algorithms and proved their correctness.

Xanadu [11] is one of first attempts to bring dependent typing into imperative languages. Notable feature of Xanadu is possibility of altering variable type during evaluation.

Deputy [12] provides flexible type system for low-level imperative languages. Types for local variables are automatically inferred with dependent typing rules. Mutable variables are supported, with type invalidation issue handled using Hoare rule of assignment. Relevant types that are affected by modification are rechecked verify well-typedness. Expressions in dependent types are restricted to local variables only. There are some cases that are not coverable by Deputy type system. In such cases Deputy augments program with assertions inserted in places where type system is unable to ensure correctness.

Goal of HTT and Ynot is to provide means for new software to rely on to be analysable using existing methods developed for functional programming. They do not themself provide any source-code analysis of software. In our approach, existing software is analysed using depend types to ensure correctness in terms of general resource management protocol.

Our system differs from Deputy (and Xanadu) in several notable ways. Mainly, these systems target at detecting memory-based defects and do not allow other resources to be verified. Furthermore, Deputy has only local type inference and variables with dependent types cannot be transferred outside of function scope. Finally, we believe that if path of static analysis is chosen for verifying program correctness, tool should no longer rely on dynamic assertions in a programs. Presence of such assertions itself opens a path for different vulnerabilities, effectively causing denial-of-service attack.

III. Overview

To identify potential flaws in software, it is not enough to solely assign some markers to code. Rules concerning transfer and modification must be defined. To represent a program we use Static single assignment form (SSA)[13]. In this form variables can be assigned only once so their binding with resource can not be invalidated.

For tracking resource state and related handle we assign markers to involved variables using special types in type system. These types not only identify type of the resource, but also include indexes, discriminating usage of methods not applicable to current state. Such indexed are called phantom [14].

While handle value can not be modified in SSA-form, type of handle variable can change phantom part of the type. This allows to restrict uses of resource handle and matches semantics of imperative languages, because previous type is no longer available for use. It is essential to provide alias creation mechanism, because we need to ensure consistency across multiple copies of a handle.

Fig. 1 show some examples of resource management operations that need certain restrictive properties to be established before operation can be performed. Possible operations include, but not limited to, creation and destruction of a resource, execution of some query, or writing to or from a resource. In case of file or network connection, there is no preliminary restriction for creating such resource. But for destruction to be correct, it must be ensured that there is no active alias that can refer to this resource after it is deallocated. Same principle applies to memory as resource. Depending on desired effect, write operation to file may be restricted to ensure exclusive access to ensure integrity of data written or stay unrestricted to allow interleaving of stream data. For memory resource, such granularity is too coarse. Per-element is possible to achieve by storing memory size in dependent type[15].

File / Network connection	
Creation	No restriction
Destruction	If have no alias
Writing	No restriction / exclusive access
Memory	
Creation	No restriction
Destruction	If have no alias
Writing	Restricted on other level
Escaped sequence	
Creation	Escaping rules must be applied
Destruction	No restriction
Execution	Safe if have resource type
Confidential Data	
Creation	Safe container marker
Destruction	Data erasure before termination
Writing	To permitted channels only

Fig. 1. Different kinds of resources

Interesting kinds of resources are escaped sequences such as SQL statements, URL's and etc., which have constraints not on destruction but on creation, requiring escaping to be performed on input from untrusted source. This guarantees that query execution will not lead to injection attack. This effectively implements basis for type-based static taint analysis[16].

Another example of resource type is confidential data. It has restrictions both on destruction and use. Destruction requires that stored data will be eased appropriately, preventing potential read in future by direct access to previously deallocated memory. Confidential data is also participating in taint analysis, where we ensure it cannot be tainted by use of disallowed data sink. Unfortunately, appropriate tainting constraints can not be automatically derived, preventing inference of this resource type.

IV. Method description

In this section we describe principle our localisation method. Localisation is carried out in three steps. First, analysed program is translated into intermediate SSA-representation[15]. Next, types of local variables are inferred from their declarations and constraints are derived by propagation of appropriate type according to set of rules. All rules are designed to be applicable if and only if corresponding operation is safe. If specific rule is not applicable, corresponding operation in unsafe in current context and will introduce software defect. If all rules applied successfully to a program, we say that program is correct with respect

$$\frac{v \leftarrow \mathbf{open}(f)}{\Gamma \vdash v : \mathbf{R}\,\mathbf{N}\,1}\ (\textsc{open})$$

$$\frac{\Gamma \vdash v : \mathbf{R}\,v_0 i \qquad v' \leftarrow v}{\Gamma \vdash v : \mathbf{R}\,v_0(i+1), v' : \mathbf{R}\,v1}\ (\textsc{hcopy})$$

$$\frac{\Gamma \vdash v : \mathbf{R}\,v_0 i \qquad v' \leftarrow \mathbf{dup}\,v}{\Gamma \vdash v' : \mathbf{R}\,N1}\ (\textsc{copy})$$

$$\frac{\Gamma \vdash v : \mathbf{R}\,v_0(i+1) \qquad \Gamma \vdash v' : \mathbf{R}\,v1}{\Gamma \vdash v : \mathbf{R}\,v_0 i}\ (\textsc{scope})$$

$$\frac{\Gamma \vdash v : \mathbf{R}\,\mathbf{N}\,1 \qquad \mathbf{close}(v)}{\Gamma \vdash v : \mathbf{Null}}\ (\textsc{scope})$$

Fig. 2. Handle typing rules

to resource management. In case of error, code location is traced back from SSA-form to original statements and user is presented with error message describing unsatisfied constraints for expected rule.

A. Basic resource management

We introduce resource type $\mathbf{R} : \tau \to \mathbf{int} \to *$ to language type system. First index of $\mathbf{R}$ tracks original resource handle for alias declarations and contains name of original variable v or special marker $\mathbf{N}$ denoting original handle. Second index is responsible for resource deallocation consistency and corresponds to number of active handle aliases.

Handle type is determined using inference rules (Fig. 2).

Rule (OPEN) is applied on resource creation. This results fresh resource handle, containing no references to other handle ($\mathbf{N}$ marker) and exist in one copy.

Rule (HCOPY) corresponds to handle copy without copying resource itself, creating an alias. This increments alias counter in original variable. Because new alias itself can serve as primary for new aliases, its alias counter starts at one.

Rule (COPY) is used when resource is fully copied and new handle is created. This corresponds to primitives such as POSIX **dup** and **accept**. Despite resources can still share some data, new handles are unrelated in respect to resource management protocol, i.e. both must be deallocated.

Most of errors in resource management are resulted by incorrect resource deallocation and access attempts after deallocation was performed. For effective analysis by type system it must first: ensure no resource alias exists after resource was closed and second: closed handle is unavailable for future use.

To guarantee absence of resource aliases after deallocation, we require that all aliases are properly nested. If this requirement is held, to close a handle all aliases must be removed in reversed order of creation. Although this ensures correctness of deallocation, it could be hard to achieve such requirement. This matters the most while analysing existing programs, because it is not possible to influence established program structure. This is why rule (SCOPE) allows some deviation of this requirement, allowing independent termination of disjoined aliases. When variable containing a handle to resource goes out of scope, parent handle alias counter is decremented, provided here are no descendant aliases from erased variable. After successful termination of all aliases, for handle with no referenced parent handle (first type index equals $\mathbf{N}$) deallocation function becomes available according to rule (CLOSE). After that resource type is set to $\mathbf{Null}$, effectively blocking all further actions. It must be stressed out that no resource leak is possible in this system because there is no rule that allows to drop a handle away without conforming to requirement of having no aliases.

Presented rules provide program correctness with respect to basic resource acquisition protocol. To use system different kinds of resources one must additionally mark each resource type with unique index, identifying type of the resource.

If additional restrictions are required, such as exclusive access, needed for ensuring integrity of file operations, more rules may be developed prohibiting write operations based on current state of alias counter.

B. Complex resource management

Complex resource management is involved when there is conditional calls to management procedures. This necessarily arises when error handling is involved. Common programming pattern for error handling is early return from procedure. This approach is error-prone, because during modification of program new resource allocations will be introduced. Existing static analysers can not diagnose such or similar behavior, because condition check may be performed by programmer at arbitrary depth of nested function calls requiring that all conditional combinations must be exhaustively verified. Basic typing rules are not expressive enough to represent properties needed to decide on availability of particular resource. To allow more detailed and precise specification of constraints, dependent types must be used (Fig. 3).

Core of idea of dependent typing here is represented by Φ-type, standing for dependent pair. It represents fixed choice made by conditional statements by saving at type level the variable used to make a choice. It should be noted that value of that variable is not saved, because there is no way to know it in advance. Neither reference to it is stored. All analysis is taking place in purely symbolic space, i.e. by name (with ambiguity removed by proper nesting of scopes). By using SSA-form we ensure that decision variable is not modified by the moment it reaches next condition evaluation. If resource deallocation was performed under condition, that condition must be verified again after different branches converge, otherwise there can be attempt to double-free a resource or attempt to skip resource deallocation and introduce memory leak.

As variables of resource type could not be lost freely by going out of scope, Φ-type containing resource in any part of expression also can not be erased. There is no way to extract condition value from Φ-type. First because it is unknown at type checking phase. Second if because there is no statements in original languages to access such extended types. Only way to retrieve resource type from conditional type is by performing condition on same variable again (rule (IF)). When appropriate condition is inserted in typing context Γ, allowing to one of the rules (Φ_{π_1}) or (Φ_{π_2}) be applied in each block under the condition. Retrieved resources then must

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2 :: *} \ (\text{PAIR}) \qquad \frac{\begin{array}{c}\Gamma \vdash b : \mathbf{bool} \\ \Gamma \vdash \tau_1 :: \kappa_1 \\ \Gamma \vdash \tau_2 :: \kappa_2\end{array}}{\Gamma \vdash \Phi(b, \tau_1, \tau_2) :: *} \ (\Phi)$$

$$\frac{\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2}{\Gamma \vdash v_1 : \tau_1} \ (\pi_1) \qquad \frac{\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2}{\Gamma \vdash v_2 : \tau_2} \ (\pi_2)$$

$$\frac{\Gamma \vdash v : \Phi(\mathbf{T}, \tau_1, \tau_2)}{\Gamma \vdash v : \tau_1} \ (\Phi\pi_1) \qquad \frac{\Gamma \vdash v : \Phi(\mathbf{F}, \tau_1, \tau_2)}{\Gamma \vdash v : \tau_2} \ (\Phi\pi_2)$$

$$\frac{\begin{array}{c}\Gamma, v_1 : \tau_1, \dots, v_n : \tau_n \vdash v_r : \tau \\ v \leftarrow \mathbf{fun}(v_1, \dots, v_n) \ \{s; \mathbf{return}\, v_r\}\end{array}}{\Gamma \vdash v : \tau_1 \to \cdots \to \tau_n \to \tau} \ (\text{FUN})$$

$$\frac{\begin{array}{cc}\Gamma \vdash b : \mathbf{bool} & \\ \Gamma, b \vdash e_1 : \tau_1 & \mathbf{if}\, b\ \mathbf{then} \ \ s_1; v_1 \leftarrow e_1; s_1' \\ \Gamma, \mathbf{not}\, b \vdash e_2 : \tau_2 & \mathbf{else} \ \ s_2; v_2 \leftarrow e_2; s_2'\end{array}}{\Gamma \vdash \varphi(b, v_1, v_2) : \Phi(b, \tau_1, \tau_2)} \ (\text{IF})$$

$$\frac{\Gamma \vdash e : \tau \quad v \leftarrow e}{\Gamma \vdash v : \tau} \ (\text{VAR-ASSIGN})$$

$$\frac{\Gamma \vdash v : \Phi(b, \tau, \tau)}{\Gamma \vdash v : \tau} \ (\text{CONVERGE})$$

Fig. 3. Dependent typing rules

be managed to converge by successfully deallocating them and establishing same type to (CONVERGE) or they will revert back to conditional Φ-type.

C. Analysis localisation

By using bidirectional propagation of types and lexically bound variable it types locality of analysis is provided. Existing dependently-typed languages suffer from fact that conditional expression may be performed with same logic by using same expressions in other order or by using another set of variables with restriction making it equivalent to former expression. To be usable by users, these system must perform combinatorial search to identify equivalence status of both expressions. This is not always possible when complex computations are involved. Moreover, it means that almost all language features must be lifted to type level with a danger for type system to become Turing-complete and thus undecidable.

In our approach we require conditions to be evaluated using exactly the same variable that was used to originally evaluate conditional statement. Otherwise types will diverge with even more Φ-types being nested. Although some flexibility is lost, it eliminates whole equivalence search procedure, allowing all expressions contributing to final choice value be hidden by single conditional variable. This variable is the only thing exported by function return type, which stays the same regardless of arguments provided. Such functional type serve as basis for information hiding which enables locality of analysis.

Bidirectional application of typing rules also contributes to minimisation of unnecessary type evaluations by bringing some constraints to call cite to provide user with more meaningful information on allowed uses of a function.

$$\frac{\Gamma \vdash v : \mathbf{RN}1 \quad \Gamma \vdash f : \mathbf{RN}1 \to \mathbf{Null}}{\Gamma \vdash f(v) : \mathbf{Null}, v : \mathbf{Null}} \ (\text{TRANSFER})$$

Fig. 4. Ownership transfer

CWE-676	Use of Potentially Dangerous Function
CWE-754	Improper Check for Unusual or Exceptional Conditions
CWE-841	Improper Enforcement of Behavioral Workflow
CWE-772	Missing Release of Resource after Effective Lifetime
CWE-456	Missing Initialization

Fig. 5. Detected weaknesses

D. Ownership transfer

As stated earlier, without special rule for transferring ownership of resource to subsequent procedures no analysis of real software will be possible, because alias counter will increment with each call made. Rule (TRANSFER) (Fig. 4) allows to transfer ownership in calls to other function if there are no aliases held to resource. It is then called function responsibility to free resource according to the rules. Obviously, ownership transfer may be performed only once in given function, because resource type is invalidated till the end of current function scope. Any attempts to use such null'ed variable will result immediate errors in type checker.

V. Detected weaknesses

Fig. 5 shows some of defects according to Common Weakness Enumeration (CWE) that can be detected by our approach. These weaknesses are not uncommon and open path for vulnerability to occur. By using static analysis, we can enforce that all boundary and special conditions are handled appropriately by programmer and themselves did not introduce new defects. Missing initialisations are detected by assigning special **Null** type by uninitialised variables. If variable will be overwritten later by initialised value, it will have corresponding type. If some execution path left that uses of potentially uninitialised variable, when error is raised by type checker.

Detection of potential resource leakage prevents many types of denial-of-service attacks. With special sensitive data markers degree of protection against direct memory access can be increased. By disabling alias creation for some resources, certain level of integrity can be established.

VI. Future work

Main direction of improvement is introducing parametric polymorphism to dependently-typed functions. This would allow to use argument preconditions to refine resulted type be eliminating unreachable combinations on particular input values and carry such proof to more deeply nested functions. It would also minimise border cases where resource usage is considered correct by programmer but such behavior is not guaranteed by observable procedure contracts.

Next, including full power of linear logic in resource type system can rise a bar for diagnosing several problems such as automatically determining what resource type can be copied freely or a limited number of times, and resources that are unique in a sense that no copies should exist.

VII. Conclusion

With presented approach it is possible to localise unsafe use of resource management operations. System is two-fold, serving not only as basis for analysis tooling, but also as guideline for minimisation resource related errors. Method applies not only to convenient resources such as files, program memory, network and database connections but also SQL statements, URL's and other entities that need input filtering. Use of typed code model makes it possible to analyse program at symbolic level while maintaining logical structure and mathematical correctness. Method is implementable as compiler extension of target language or as separate analysis tool. Possibility of static analysis gives opportunity to locate resource management defects in existing software. By localisation and correction of defect causes software quality in increased and potential vulnerabilities are eliminated.

References

[1] O. Kiselyov and C.-c. Shan, "Lightweight monadic regions," in *ACM Sigplan Notices*, vol. 44, no. 2. ACM, 2008, pp. 1–12.

[2] B. Stroustrup, "Foundations of c++," in *European Symposium on Programming*. Springer, 2012, pp. 1–25.

[3] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.

[4] U. Norell, "Dependently typed programming in Agda," in *Advanced Functional Programming*. Springer, 2009, pp. 230–266.

[5] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 05, pp. 552–593, 2013.

[6] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[9] A. Nanevski, G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *Journal of Functional Programming*, vol. 18, no. 5-6, pp. 865–911, 2008.

[10] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal, "Ynot: dependent types for imperative programs," in *ACM Sigplan Notices*, vol. 43, no. 9. ACM, 2008, pp. 229–240.

[11] H. Xi, "Imperative programming with dependent types," in *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*. IEEE, 2000, pp. 375–387.

[12] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Programming Languages and Systems*. Springer, 2007, pp. 520–535.

[13] C. Wimmer and M. Franz, "Linear scan register allocation on ssa form," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 170–179.

[14] J. Cheney and R. Hinze, "Phantom types (2003)."

[15] L. Tsvetkov and A. Spivak, "Utilizing type systems for static vulnerability analysis," in *Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT), 2016 18th Conference of*. IEEE, 2016, pp. 345–350.

[16] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2014, pp. 140–154.