Utilizing Type Systems for Static Vulnerability Analysis

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

Abstract—**Programming languages use type systems to reduce number of bugs. Type systems of most languages are not powerful enough to express basic exception safety. Extension of type system in a way that allows representing exception guaranties can provide valuable information to analysis tools. Such tools could even be implemented in type system. We describe a way to extend type system of a given language allowing security invariants to be expressed and vulnerable code to be located.**

I. Introduction

Static vulnerability analysis is a form of static source code analysis with primary goal of detecting software defects causing potential security breach. Static analysis is a preferable method for security assurance because attacks often targeted at rarely executed code paths witch are not covered by test cases and with static analysis it is possible to cover all of them.

Many programming languages have mechanism of preventing ill-formed programs to be compiled and executed — type system. Each term in a language is given corresponding "type" serving as marker for kind of value stored in a variable or allowed as function argument. Type annotations restrict operations done on variable, assuring they are sensible and conform to language semantics. Language is called typesafe if where is no possibility to violate types in run-time [1]. For this reason static type checking can be considered as form of static program analysis.

Ability of rich type systems to eliminate whole classes of errors was noted long time ago and was developed extensively in functional languages, most notably Haskell, where monads are of great use for limiting side effects of computation [2]. It is common knowledge among Haskell community that type system provides trustworthy approach to restricting program behavior and enforcing correctness not only operationally but also semantically. Languages like Agda [3] and Idris [4] take this further, allowing types to depend on values, that is, they provide dependent typing. This allows types to be more precise and target properties of program you want to establish or verify.

Type systems for imperative languages do not have such success. Recently attempt was made to bring modern type system to imperative world — Rust programming language [5]. Notable feature of Rust is borrow checker [6]. Rust implements feature called "borrowing" — it allows to have many read-only references point to same variable, but prevents their use if mutable reference was taken. Variable is then said to be "borrowed". Such rules implement what is known as affine type system and provide strict solution to aliasing problem.

Augmenting existing languages with more elaborative type systems can provide valuable insights on program structure, deduce relations between variables and functions, as well as their interaction patterns. This information can be used to locate and possibly exploit vulnerabilities present in analysed software.

In this paper we present a model of locating memory-based vulnerabilities for imperative languages in presence of mutable state and side effects using type system with dependent types. Following issues are addressed in this paper:

- Mutability: imperative languages encourage use of mutable variables. If variable is assigned a type at point of declaration, subsequent assignments may invalidate proofs referencing this variable. Previous works either disallow changing established types [7] or take approach of tracking types affected [8].
- Linearity: improper usage of memory allocation functions has always been a source of memory leakage and corruption, violating data integrity. Wherefore for effective vulnerability detections system it is crucial to have mechanism for tracking such functions. Model of linear resource could be expanded to ownership model and notion of protocol for high-level languages.
- Aliasing: pointer aliasing in presence of linearity might produce type system incapable of representing many simple programs that utilize aliasing. Care is needed to track aliased variables or highly restrict possible aliasing.
- Reproducibility: if types can not be unified there should exist input that violates determined requirements. System should be then provide user with test case that allows malformed data to sneak into located place.

First, in Section II existing works in this area are examined to give reader a grasp of subject. In Section III we present general approaches and challenges of type-based vulnerability analysis and sketch solutions to them. Specifically, Section III-A and III-C describe methods used to integrate mutability inside analysis framework without interfering with type system. Section IV describes intermediate language structure, types and inference rules used to support automated program analysis. Particularly, Section IV-A describes internal representation structure and ways for transforming existing programs into it. Section IV-B explains types that are assigned to variables for carrying determined properties. Section IV-C describe interaction patterns that lead to inferring desired properties about memory access, such as ensuring no buffer overruns. In Section V some insight is discussed on generating exploits to found defects that can be used by end-users to determine defect threat level and apply means to fix it's causes. Finally, directions of future work are outlined in Section VI.

II. Related works

Related works targeted at detection of memory-based errors can be categorised to dynamic approaches and static approaches.

CETS [9] applies compile-time instrumentation to pointer operations to detect dangling pointers. Pointers are augmented with metadata, stored independently of original pointer contents. When load-store operations are performed, metadata is accessed to verify that pointer is still allocated. CETS can detect dangling pointer access on heap as well as on stack, even if memory was free'd and allocated again in same place. According to authors, slowdown ranges from 48% to 116% total.

Address Sanitizer [10] takes shadow memory instrumentation approach to diagnose out-of-bounds access to global, local and heap objects. It can also detect use-after-free errors. Detection of violations of stack and global access is possible through red zones added between consecutive objects. With compressed shadow state encoding, average slowdown of Address Sanitizer is 73% with no false-positives.

Both Address Sanitizer and CETS are very effective systems to locate memory errors, but full path coverage is needed to ensure memory-correctness. For large software this would imply rather big testing suit. Although slowdown is of less concern on modern systems, it presents problems for responsive systems.

For static memory analysis methods, several papers were published with intent to incorporate dependent types and imperative languages.

Hoare Type Theory [11] proposes system where Hoare Logic is embedded into type system. Imperative computations are executed in indexed Hoare monad. Dependent typing is allowed on Hoare triples so postconditions can depend on returned value. HTT uses two-stage type checking to overcome undecidability issue.

Ynot [12] is next step after HTT. It is implementation of HTT in language of Coq proof assistant. Ynot allows imperative programs with side-effects be implemented in pure functional languages. HTT dependent type features are already provided by Coq. With this approach authors have successfully implemented several imperative algorithms and proven their correctness.

Xanadu [7] 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 [8] provides flexible type system for low-level imperative languages. Dependent types could be automatically inferred for local variables. Deputy also supports mutable variables. Type invalidation issue is handled using Hoare rule of assignment. Track is kept for affected types to verify well-typedness. Expressions in dependent types are restricted to local variables only. Assertions are inserted in augmented program to find errors not covered by typing system.

HTT and Ynot both serve as evaluation models of imperative programs. They do not themself provide any source-code analysis of software. We concentrated on augmenting existing program with dependent types to locate potential vulnerabilities.

Our system differs from Deputy (and Xanadu) is several notable ways. First, neither of this systems allow negative array access, which is possible in presence of pointer arithmetic available on low-level languages. Second, memory allocation issues are not addressed — Deputy relies on correct malloc/free usage. We outline that vulnerability prevention system shall provide complex memory safety guaranties. Furthermore, Deputy has only local type inference and variables with dependent cannot propagate though function return value. Finally, we believe that for applications requiring high service continuity, static assurance of correctness is preferred, with redundancy measures to cover critical software or hardware failure. With this in mind, programs augmented with assertions do not provide level of security expected from system based on logic expressions.

III. Overview

Types provide solid and exhaustive approach to path coverage and variable constraints with polymorphic typing. Dependent types expand this further, providing a way to distinguish assertions to be made when several execution path are joined together. Moreover, in low-level programming, actions available to perform on variable are dictated not only by it's value, but values of other variables in scope — array bounds, union tags and, of course, function arguments. Ability to construct and propagate proposition over function arguments is main advantage of dependent types for static analysis — they can conceal function implementation, allowing only external behavior to be visible to type checker and user. This allows modular inter-procedural analysis of software required for successful vulnerability detection, as many hard-to-find bugs are result of wrong function interactions.

Main goal of this paper is to demonstrate possibility of locating memory-based defects and vulnerabilities in software developed in high-level languages by inferring properties and constraints on variables and functions with no assistance from user and requiring no extra code annotation or binary instrumentation.

With support from compiler it is possible to include additional type checking stage performing security analysis. Found defect could be exploited to determine severity of damage that could be caused to the system and data integrity.

Previous approach to make languages safer with rich type systems either involved code instrumentation [13] or required at least some annotation [8]. We consider that any annotation burden limits performance greatly, because vulnerability analysis could be run by quality assurance officer who has no complete understanding of invariants available or by security researcher who has no familiarity with code.

A. *Mutability*

While there are many functional languages that support mutable variables, imperative languages favor their use. This creates real challenge for language tools: compilers and static analysers. For type system it means that either types cannot change when established, or such modifications should be tracked and types adjusted accordingly. Several approaches were proposed to workaround this problem: restricting dependencies to constants and locals [8] or allowing type to change during execution [7].

Well-known approach exists for transforming program with mutable variables to program with immutable variables — Static single assignment form (SSA). In this form variable can be assigned only once so type can not change later. It provides convenient model and opens a way for optimizers and other analysis tools. Within SSA in different blocks of control flow graph (CFG) variables of original language have different names and can be distinguished.

To join several execution paths together, Φ-functions are used. Value of Φ-function depends on which branch of CFG it is coming from. In high-level languages branching is hierarchical so only last diverged branches can be joined together. With this in mind Φ-function can be replaced with original branching condition, because it cannot be invalidated thanks to SSA. This lands perfectly to dependent types: they itself are mechanism for distinguishing different properties varying on original value, in this case a branching condition. Dependent pairs are used to represent such situations with first argument serving as witness and second as proof of some inferred property.

Such approach means that type checking should be carried out after program call graph is build, which requires some primary (language-provided) type checking to be already performed. For example, consider language with overloadable functions, such as C++: type of argument dictates which function overload is called.

B. *Linearity*

Linear protocols and resources are extensively used in programming. Examples of such protocols are: memory [de]allocation, I/O, object initialization/destruction and other stateful commands. In all this cases life cycle is the same:

1) object is not valid
2) object initialization
3) operations can be performed
4) object destruction
5) object invalidated.

At least two distinguished states exist which determine operations to be performed. Invalid object can only be initialised and for object to be destroyed it should be valid. In this case states are linearly ordered so we can say that object is linear, that is, it follows a linear protocol.

Inability to follow such protocols results many memory leaks, crashes and data corruptions in modern software. Memory leak issue is often addressed in modern languages by introduction of garbage collector, but file I/O is still considered a different issue. RAII is good remedy for this case, but not supported in many concerned languages, such as Java and Python.

If object has many states and availability of object methods depends on particular state, there is no good solution for enforcing correct usage. With help of type-indexed or value-indexed types such states could be tracked down to point of relevant function application.

C. *Aliasing*

Pointer aliasing obstruct inference in two ways. First, if variable that was used as part of the type, e.g., Φ, modification of it can invalidate inferred properties. To overcome this issue, we only allow local variables to be included in types. SSA form generally protects local variables from modification, unless references are taken into account. We see simple remedy for this — creating two versions of stack variables. Referencing information is fully known at compile time, so we can de-alias stack variables. Reference will be taken only to one of them and can be modified. Second version can be used in types with value that variable was initialised with. This greatly simplifies inference rules, because no memory-read or memory-write rules are needed. All memory access can happen transparently to type system. Original version of variable is used to form type, and other is subject to possible modifications.

IV. MODEL DESCRIPTION

In this section we describe main principles of our analysis framework, modeled over simple imperative language. Analysis is performed in several steps. First, program being analysed is translated into intermediate language with limited instruction set. Next, types that are inferred and checked according to framework rules are introduced to language terms. In the end, if type checking succeeds, it is deemed that no out-of-bounds access or double-free can be done. Additionally, typing context could be examined for still allocated references, detecting potential memory leaks. If type checking doesn't succeed, term triggering error is traced back to original language and user can be provided with example input exploiting undefined behaviour.

A. *Language*

For analysis to take place, program should be translated to intermediate representation (IR). Using intermediate language makes framework independent from source language as well as allows additional constructions.

Intermediate representation is defined at Fig. 1. In IR no variable can be assigned more than once, i.e., it is kept in SSA form. This keeps system simple and concise, mainly because once variable is defined it has been assigned a value and corresponding type that cannot be altered hence it is safe to refer to them later while they are in scope. Because no type modification can occur, we should require three normalisation rewriting rules be applied prior to conversion to SSA:

$$\begin{aligned} v_1[v_2] &\Rightarrow v' \leftarrow v_1 + v_2; {}^*v' \\ \&v_1[v_2] &\Rightarrow v_1 + v_2 \\ \mathbf{free}\, v &\Rightarrow v ::= \mathbf{free}\, v \\ \&v &\Rightarrow v' ::= v; \&v' \end{aligned}$$

Kinds	$\kappa ::= * \mid * \to \kappa \mid \tau \to \kappa$
Types	$\tau ::= B \mid L \mid \tau_1\ \tau_2 \mid \tau\ e \mid \Phi(b,\tau_1,\tau_2) \mid \tau_1 \times \tau_2$
Built-in types	$B ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{uint} \mid \mathbf{ref}\, L\ \tau\ i\ j$
Index expr.	$i,j ::= C \mid v \mid i\,\mathbf{op}\,j \mid \varphi(b,v_1,v_2)$
Expressions	$e ::= j \mid l \mid \&l \mid e_1(e_2) \mid \mathbf{not}\, b$
L-values	$l ::= v \mid {}^*v \mid v_1[v_2] \mid \langle v_1,v_2\rangle$
Constants	$C ::= D \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{Null} \mid 0 \mid 1 \mid \dots$
Statements	$s ::= \varepsilon \mid s_1; s_2 \mid l \leftarrow e \mid \mathbf{if}\, b\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2$
Declarations	$D ::= M \mid \mathbf{fun}(v_1,\dots,v_n)\{s; \mathbf{return}\, v_r\}$
Pointer origin	$L ::= \mathbf{I} \mid \mathbf{S} \mid \mathbf{H} \mid \mathbf{D}$
Memory alloc.	$M ::= \mathbf{alloca}\, \tau j \mid \mathbf{malloc}\, \tau j \mid \mathbf{free}\, v$
	$v \in$ Variables
	$b \in$ Boolean variables
	op $\in$ Binary operators

Fig. 1. Grammar for intermediate representation

First two rules rewrite array access operator in terms of pointer arithmetic and dereference. Third rule make it possible to invalidate pointer deallocated using **free**. Function **free** is usually declared in source language as function with side-effect and no return value. Here value returned is used to invalidate pointer in question because further uses would be directed to another variable version after conversion to SSA. Last rule decouples local variables that are accessed through pointers so variables present in types cannot be aliased. This rules are essential for correct analysis and allow to simplify inference without loosing generality.

Program in IR consists of (global) declarations D, statements s and expressions e. Following C tradition function declarations are considered constants and cannot be created at run-time or compile-time. Statements are either sequential, branching or variable assignment. Statements do not have type assigned to them. Instead, declared variables form typing context Γ.

Existing programs can be transformed into this internal representation using SSA algorithms implemented in many modern compilers [14]. Internal representation is not tied to particular language, so approach taken can be extrapolated on many imperative languages.

B. Types

There are several types present in the system. Type of relational operators is **bool**, type of integer expressions is **int**. Type **uint** is restricted to non-negative integers. There also is a pair type that corresponds to structure definition (composite type). Pair type is essential to successful inference because it provides the only way to return witness variable from a function.

$\mathbf{I}, \mathbf{S}, \mathbf{H}, \mathbf{D}$ are themself a concrete uninhabited types of kind $*$ and serve only as markers for **ref** type.

Most notable type is **ref** type. It is type of arrays, pointers, lists and etc. It is indexed with two types and two expressions. First index referred as L keeps track of origin of pointer: $\mathbf{I}$ for invalid pointers, $\mathbf{S}$ for pointers to stack variables, $\mathbf{H}$ for objects on the heap and $\mathbf{D}$ for pointers that are detached from their original source. This index is determined by function that created such pointer and required to correctly track it to deallocation routine, preventing memory leaks. Second type index τ identifies type of elements that reference is referring to. Finally, two index expressions i, j specify how many elements available for access though this pointer. Fig. 2 shows accessible range corresponding to some reference. Range is split into positive and negative indexes to allow pointer to advance back and forth freely without losing information. Altering pointer position is a common idiom in C language so keeping track of items available behind pointer is needed. Note that pointer can refer to element within closed interval determined by $[-i, j]$, but only elements of half-closed range $[-i, j)$ can be accessed, i.e., dereferenced with * operator.

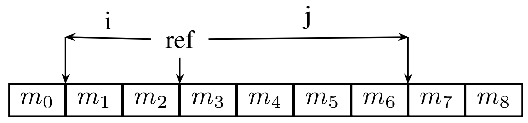

Fig. 2. Structure of reference type

Second notable type that system is build upon is Φ-type. It can be seen as dependent pair type with second element fixed to choice operator between types τ_1 and τ_2 witnessed by first element of type **bool**. It is type of φ-function and only source of this type is branching statement. As noted earlier, due to immutable nature of variables present in SSA and hierarchical structure of conditional statements, resulting type of φ-function can be reconstructed using variable on which choice was made.

There is also a function type that is assigned to pre-defined functions and functions defined before type checking stage. Main concern about functions is possibility of information hiding. This rises a question whenever analysis can inspect whole functions and expose internal structure of it should be limited to observable behavior only.

C. Rules

In this section we present typing rules for our intermediate representation language described in Section IV-A. Core typing rules are show on Fig. 3. This includes rules for types indexed by other types and expressions, kinds of built-in types, pairs, Φ-pairs and their projections. Some typing rules, such as types for constants and binary operators, are omitted for clarity. Fig. 4 shows rules for creating and manipulating references.

Specifically, rule (ADDR-OFF) creates a reference for local stack variable v that obviously can refer to one element only. Rule (DEREF) is main rule for accessing array elements. To use this rule it must be known somehow that pointer has at least one element in front of it. This fact is trivially satisfiable if pointer was originally declared within scope or used in loop with array length serving as limiter. If it is not there, some variable carrying proof of available elements must be passed from pointer origin. Such variable or constant should exist, otherwise neither programmer nor analysis tool can perform safe pointer dereference.

$$\frac{\Gamma \vdash \tau_1 :: \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \tau_2 :: \kappa_1}{\Gamma \vdash \tau_1 \tau_2 :: \kappa_2}\ (\text{TYP TYP})$$

$$\frac{\Gamma \vdash \tau :: \tau' \to \kappa \qquad \Gamma \vdash e : \tau'}{\Gamma \vdash \tau\, e :: \kappa}\ (\text{TYP EXP})$$

$$\frac{\begin{array}{ll} \Gamma \vdash \tau :: * & \Gamma \vdash i : \mathbf{uint} \\ \Gamma \vdash L :: * & \Gamma \vdash j : \mathbf{uint} \end{array}}{\Gamma \vdash \mathbf{ref}\, L\ \tau\ i\ j :: *}\ (\text{REF TYP})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ (\text{APP})$$

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

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

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

Fig. 3. Basic kinding and typing rules

$$\frac{}{\Gamma, \tau :: \kappa \vdash \mathbf{Null} : \mathbf{ref}\, \mathbf{I}\ \tau\ 0\ 0}\ (\text{NULL})$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \&v : \mathbf{ref}\, \mathbf{S}\ \tau\ 0\ 1}\ (\text{ADDR-OF})$$

$$\frac{\Gamma \vdash v : \mathbf{ref}\, L\ \tau\ i\ j \qquad \Gamma \vdash v_p : j \neq 0}{\Gamma \vdash {}^*v : \tau}\ (\text{DEREF})$$

$$\frac{\Gamma \vdash v : \mathbf{ref}\, L\ \tau\ i\ j}{\Gamma \vdash \&^*v : \mathbf{ref}\, \mathbf{D}\ \tau\ i\ j}\ (\text{DEREF-ID})$$

$$\frac{\begin{array}{rr} & \Gamma \vdash v_2 : \mathbf{int} \\ \Gamma \vdash v_1 : \mathbf{ref}\, L\ \tau\ i\ j & \Gamma \vdash v_p : v_2 \in [-i, j] \end{array}}{\Gamma \vdash v_1 \pm v_2 : \mathbf{ref}\, \mathbf{D}\ \tau\ (i \pm n)\ (j \mp n)}\ (\text{PTR-ARITH})$$

Fig. 4. Reference typing rules

Rule (PTR-ARITH) alters type of reference when pointer is advanced it either direction. Again, where should exist some variable in scope to prove that index is in allowed range. Rule (DEREF-ID) allows to re-take reference for one-past-last element of array. Such reference would be otherwise inaccessible after dereference because such dereference is forbidden by (DEREF) rule. Note that (PTR-ARITH) and (DEREF) rules are used for element access even with $v_1[v_2]$ operator due to rewriting rules described in Section IV-A. Also note how both rules mark resulting pointer "detached". If this was not the case, creation of multiple pointers having same value and origin markers would be possible. This would lead to possible double-free problems and corrupted heap.

Final set of rules on Fig. 5 describes how variables inside

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

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

$$\frac{\Gamma \vdash e : \tau \qquad v \leftarrow e}{\Gamma \vdash v : \tau}\ (\text{VAR-ASSIGN}) \qquad \frac{\begin{array}{r} \Gamma \vdash j : \mathbf{uint} \\ \Gamma \vdash \tau :: * \\ v \leftarrow \mathbf{alloca}\, \tau j \end{array}}{\Gamma \vdash v : \mathbf{ref}\, \mathbf{S}\ \tau\ 0\ j}$$

$$\frac{\begin{array}{r} \Gamma \vdash j : \mathbf{uint} \\ \Gamma \vdash \tau :: * \\ v \leftarrow \mathbf{malloc}\, \tau j \end{array}}{\Gamma \vdash v : \mathbf{ref}\, \mathbf{H}\ \tau\ 0\ j} \qquad \frac{\begin{array}{c} \Gamma \vdash v : \mathbf{ref}\, \mathbf{H}\ \tau\ 0\ j \\ v' \leftarrow \mathbf{free}\, v \end{array}}{\Gamma \vdash v' : \mathbf{ref}\, \mathbf{I}\ \tau\ 0\ 0}$$

Fig. 5. Typing rules inside statements

statements are typed. According to rule (FUN), type of function should be determined by type of arguments. This essentially requires that variables that participate in returned type must originate either from argument or returned within resulting composite variable so they cannot go out of scope. Rule (VAR-ASSIGN) simply tell us that type of new variable is type of original expression. Most important rule here is (IF). Depending on condition, typing context is expanded with proof of b and then evaluating type of e_1 or with proof that b is not the case and evaluating type of e_2. Resulting variables are then combined to Φ-functions and types to be carried along to point of usage. φ-function first argument and Φ-type index are restricted to concrete variables, so that type could not be invalidated later. To extract type stored in Φ projections $\Phi\pi_1$ and $\Phi\pi_2$ are used automatically when condition was verified and present in context.

Finally, types of memory allocation functions are given. **alloca** for stack allocation, **malloc** for heap allocation. References are given corresponding marker to ensure that no incorrect pointer would be passed to deallocation function **free**. **free** return value is used to invalidate passed pointer for next usage because of rewriting rules applied prior to conversion to SSA. Marker **I** is present only for convenience: inability to use invalid pointer comes from having both indexes reset to zero, no dereference can take place according to rule (DEREF). More over, with marker reset to invalid, this pointer can no longer be free'd, preventing heap corruption.

D. Inference

Type inference is generally guided using rules described in previous subsection. However, recursive functions can prevent analysis from complete type-checking. Recursive functions can come from original language or be generated as part of SSA stage. In either case they are one of the obstacles to exhaustive program analysis. To correctly determine function type it must be terminating. This property could not be determined for every function due to Halting Problem.

One workaround of this limitation can be made as follows: note that recursive call is done either conditionally or unconditionally. If it is unconditional, function does not terminate and type could not be determined. If it is conditional, value returned should descend from Φ-node. In this case call to such function can be type-checked up to Φ-type until specific branch is proven to execute.

V. Exploiting vulnerabilities

With help of type system, potential software defects can be identified. But it remains unclear whenever found defect can be exploited. There can be three results of type checking: all types are correct and traceable by both analysis and programmer; definite error — contradiction was found during analysis; and unification error due to absence of complete information. In case of definite error, variables can be traced up to their origin. During this process, expressions that lead to error are reconstructed (they was already present inside system, so error was found) and test case is generated. In case of error due to incomplete information, such as presence of opaque function, SMT solver should be used for variables that are know to participate in ambitious expression. Solver is needed here because our inference rules do not provide polymorphism and cannot represent unknown or quantified types. Then constrains are determined, fuzzing methods can be effectively applied. Note that usage of SMT and fuzzing is not part of checking algorithm but mere attempt to provide user with a test case so they are convinced of defect and have staring point for debugging.

VI. Future work

We plan to develop our system in three main directions. First, including full-fledged linearity into type system. With full power of linear logic type checker would have full information whenever variable or expression can be modified by later statements and memory writes. It would then allow safely use memory references in dependent types. This also includes global variables.

Second, in current setup every function call should actually be re-typed each time it is referenced. This comes from fact that monomorphic function in source language can be typed differently in intermediate representation in each new call context (e.g., more information is known statically). This can be fixed by back-propagation of constraints in functions. Currently, constrains propagate in forward direction only.

Third, it may be beneficial to include parametric polymorphism into type system and intermediate language even though source languages do not support it. This would allow us to quantify over expressions participating in dependent types, which is required for opaque functions. I.e., one could run analysis on library and derive appropriate inner- and inter-function interaction constraints and use this information separately for each program using this library. Essentially, this is form of information hiding that provides modular approach to software analysis, retaining exhaustiveness of path coverage.

VII. Conclusion

In this paper we presented simple approach to software analysis using type system. Analysis can provide insight on possible memory-based vulnerabilities, such as buffer overflow, memory leaks and corruption. Method could be applied at stage of software development, for example, on continuous integration server, as part of software quality assurance. Static checking can locate bugs in low-attention regions of code that are not covered by unit testing.

Languages with rich type systems usually follow functional programming paradigm and have many advantages over conventional imperative languages. Now existing software can benefit from additional typing schemes applied to their code to discover defects and vulnerabilities.

References

[1] R. Harper, *Practical foundations for programming languages.* Cambridge University Press, 2012.

[2] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages.* ACM, 2007, pp. 12–1.

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

[4] 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.

[5] The Rust Programming Language, Web: https://www.rust-lang.org/.

[6] E. Reed, "Patina: A formalization of the Rust programming language," *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

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

[8] 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.

[9] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

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

[11] 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.

[12] 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.

[13] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[14] *Static Single Assignment Book*, 2015, unpiblished, Web: http://ssabook.gforge.inria.fr/latest/book.pdf.