基于静态分析的安全漏洞检测技术研究*)

夏一民 罗 军 张民选
(国防科学技术大学计算机学院 长沙 410073)

摘 要 消除软件中的安全漏洞是建立安全信息系统的前提。静态分析方法可以自动地提取软件的行为信息,从而检测出软件中的安全漏洞。和其它程序分析方法相比,该方法具有自动化程度高和检测速度快的优点。在本文中,我们首先描述了静态分析的理论基础,然后说明了类型推断、数据流分析和约束分析等主要静态分析方法及其在安全漏洞检测中的应用,并比较这些方法的优缺点。最后给出了几种支持安全漏洞静态检测的编程语言。

关键词 安全漏洞,静态分析,抽象解释,类型推断,数据流分析,约束分析,信息安全

Security Vulnerability Detection Study Based on Static Analysis

XIA Yi-Min LUO Jun ZHANG Min-Xuan
(School of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract Security vulnerability of software is a serious threat for information security. Static analysis can find security vulnerabilities by automatically deriving information about the behavior of software. Comparing with other program analysis methods, static analysis method can detect security vulnerabilities automatically and effectively. This paper presents the theory basis and principles of static analysis methods, and introduces their applications and characters in security vulnerabilities detection. At last, we show some security languages which can support detection of security vulnerability.

Keywords Security vulnerability, Static analysis, Abstract interpretation, Type inference, Dataflow analysis, Constraint analysis, Information security

1 引言

软件作为现代社会的基础设施,已广泛应用于能源、交通、通信、金融和国防等安全攸关领域中,软件中的任何安全漏洞都可能导致非常严重的后果。一个安全的软件必须保证对任何输入都不会产生不期望的结果[1]。随着软件系统变得越来越复杂和庞大,软件中的安全漏洞也急剧增加,人们急需一种自动化检测方法来帮助程序员发现实际软件中的安全漏洞。目前,检测软件安全漏洞的方法主要有3种:动态测试、形式化验证和静态分析。动态测试方法通过软件的实际执行来发现软件中的错误,由于只能对有限的测试用例进行检查,所以不能保证发现软件的所有安全漏洞。形式化验证方法包括定理证明和模型检验两类,它们都可以精确地确定软件的属性。定理证明方法将程序转换为逻辑公式,然后使用公理和规则证明程序是一个合法的定理。由于定理证明过程难以完全自动化,需要高素质分析人员的大量参与,证明过程非常耗时费力,一般只用于验证设计阶段的程序规范而非实际代码。模型检验用状态迁移系统(S)描述软件的行为,用时序逻辑、计算树逻辑或 μ-演算公式(F)表示软件执行必须满足的性质,通过自动搜索S中不满足公式F的状态来发现软件中的漏洞。由于需要穷尽程序的所有实际执行状态,所以模型检验的效率很低,并且不能检验无穷状态系统。最近几年,静态分析作为一类高效的程序分析方法,受到越来越多的重视。当用户给出语言的抽象语义以后,该类方法能够自动发现满足所有可能(不一定实际存在)执行状态的软件属性。静态分析方法具有自动化程度高、分析速度快,并且可以检验无穷状态系统的优点。尽管静态分析方法可能产生一定的漏报(false negatives)或误报(false positives),但仍然是当今最实用、最有效的安全漏洞检测方法之一。

本文下面的章节是这样安排的:第2节描述静态分析的基础,第3节详细介绍类型推断、数据流分析和约束分析等主要静态分析方法及其在安全漏洞检测中的应用,第4节比较各种静态分析方法的优劣,第5节介绍近年来程序语言技术的进展。

2 理论基础

抽象解释[2](Abstract Interpretation)是静态分析的形式化框架。静态分析仅跟踪用户关心的程序属性,所以它对程序语义的解释是程序实际语义的近似。如果程序执行时的实际状态序列为 $v_0 \rightsquigarrow v_1 = f_c(v_0) \rightsquigarrow \cdots \rightsquigarrow v_i = f_c(v_0) \rightsquigarrow \cdots$,其中 $v_i \in V$,V 表示程序执行时的实际状态集合,f_c 表示实际解释函数,那么程序属性就是 f_c 的最小不动点。由于源程序的任何非平凡属性都是不可判定的[3],抽象解释设计程序语义的抽象解释函数 f_a,再用 f_a 的最小不动点来近似 f_c 的最小不动点。此时,程序的抽象执行序列为 $l_0 \mapsto l_1 = f_a(l_0) \mapsto \cdots \mapsto l_i = f_a^i(l_0) \mapsto \cdots$,其中 $l_i \in L$,L 表示抽象状态集合。对于抽象域 L 构造的完全格 $\langle L, \sqcup, \sqcap, \top, \bot, \sqsubseteq \rangle$ 抽象解释定义了实际状态和抽象状态之间的正确关系:关系 $R \subseteq V \times L$ 是

*)国家"863"高技术研发计划基金项目:重大软件专项服务器操作系统内核(2002AA1Z2101)资助。**夏一民** 博士生,主要研究领域为信息安全;**罗 军** 教授,主要研究领域为信息安全和并行程序设计;**张民选** 教授,主要研究领域为微电子和体系结构。

正确关系，当且仅当 R 满足以下两个条件(1) $\forall v\in V, \forall l_1, l_2\in L, (v\,R\,l_1)\wedge(l_1\sqsubseteq l_2)\rightarrow(v\,R\,l_2)$；(2) $\forall v\in V, \forall L'\subseteq L, (\forall l\in L', v\,R\,l)\rightarrow v\,R(\sqcap L')$。任何满足以下两个条件的静态分析都是正确的：(1) l_0 是 v_0 的安全近似，即 $v_0\,R\,l_0$；(2)每次迁移都保持正确关系 R，即 $\forall v_1, v_2\in V, \forall l_1, l_2\in L, (v_1\rightsquigarrow v_2)\wedge(v_1\,R\,l_1)\wedge(l_2=f_a(l_1))\rightarrow v_2\,R\,l_2$。

当 f_a 的不动点难以计算时，比如收敛速度太慢或者 L 存在无限上升链，抽象解释允许用户建立一个更近似的抽象域 M 和新的抽象函数 $f_m: M\rightarrow M$ 来近似原有的抽象域 L 和抽象函数 $f_a: L\rightarrow L$，但要求 f_m 满足 $f_m\sqsupseteq\alpha\circ f_c\circ\gamma$。如果新关系 $S\subseteq V\times M$ 的定义是：$v\,S\,m$ 当且仅当 $v\,R\,\gamma(m)$，那么可以证明 S 是正确的关系。当 $\langle L,\alpha,\gamma,M\rangle$ 是一个 Galois 连接时，可以证明 f_m 的每次计算都将保持正确关系 S。

Widening/narrowing 操作也可以获得与 Galois 连接方法相类似的精度和收敛速度，并且更加通用。也就是说，即使不能找到将无限域 L 抽象到有限域的 Galois 连接，也一定可以找到合适的 widening/narrowing 操作，保证抽象解释函数的快速收敛。Widening 和 narrowing 的操作函数分别记作 $\nabla: L\times L\rightarrow L$ 和 $\nabla: L\times L\rightarrow L$。给定完全格 L 上的单调函数 $f: L\rightarrow L$，如果定义函数 $f^n_\nabla=f^{n-1}_\nabla\nabla f(f^{n-1}_\nabla)$，且 $f^0_\nabla=\perp$，那么 f^n_∇ 一定存在一个最小不动点 f^m_∇；如果定义函数 $f^n_\nabla=f^{n-1}_\nabla\Delta f(f^{n-1}_\Delta)$，且 $f^0_\nabla=f^m_\nabla$，那么 f^n_∇ 也存在最小不动点，并且序列 $(f^n_\nabla)_n$ 和 $(f^n_\nabla)_n$ 的元素都是 $lfp(f)$ 的安全近似，但 $(f^n_\nabla)_n$ 比 $(f^n_\nabla)_n$ 更加精确。

3 静态分析方法

虽然抽象解释给出了正确设计静态分析的充分条件，但用户还需要根据被检测安全漏洞设计具体的抽象解释函数和抽象域。目前，静态分析主要有类型推断、数据流分析和约束分析 3 种方法，这些方法都可以看成是抽象解释的实例。

3.1 类型推断

一个类型表示一个值的集合。根据变量的安全属性设置变量的类型，就可通过类型推断和类型检查发现违反安全规范的操作。一个类型系统由类型语法、静态语义和动态语义等几个部分组成，静态漏洞检测主要关心类型静态语义中的定型断言、推断规则和检查规则。定型断言(typing assertion) $\Gamma\vdash M: T$ 表示变量 M 在静态定型环境 Γ 中具有类型 T，推断规则 $\frac{A}{B}$ 表示从前提 A 可以推导出结论 B，检查规则用于判定某个操作是否是良行为的(well behaved)。此外，由于应用不同的推断规则可能推导出同一个表达式的不同类型，因此还需要一个类型推断算法确定推断规则的应用方式。多数类型系统都是流不敏感和上下文不敏感的，即每个变量在整个程序中只能有一个类型。上下文敏感的类型推断可以根据函数调用点的上下文确定函数和函数内变量的类型(称这种具有多个类型的函数为多态函数)，而流敏感的类型推断允许变量在不同的程序点具有不同的类型[4]。类型之间可以构成子类型关系：类型 B 是类型 A 的子类型，记为 $B<A$，表示 B 类型的变量可以在任何时候替换 A 类型的变量，反之则不可。

基于类型推断的安全漏洞检测常使用类型限定词来扩展源语言的类型系统[4]，而不必构造一个全新的类型系统，使得新类型系统既兼容原有的类型系统，又减少了工作量。类型限定词将变量类型表示的值集合划分为几个互不相交的子集(子类型)，分别表示不同的安全级别。静态检测时，先由用户指定少数变量的类型限定词，然后应用类型推断算法进行类型推断，并对推断结果进行类型检查，从而发现违反安全规则的敏感操作。

类型推断方法具有简单、高效的特点，非常适合软件安全漏洞的快速检测。已有的采用类型推断方法检测的安全漏洞主要有 C 程序中的格式化字符串漏洞[5]、操作系统内核中的权限检查[6]，以及操作系统内核中不安全的指针使用[7]等。

3.2 数据流分析

数据流分析是一种在编译优化领域得到广泛应用的程序分析方法。与类型推断仅仅考虑变量的类型不同，数据流分析可以检查变量的任意属性，包括变量的到达定义、变量的取值范围等等，从而可以建立更加复杂的分析状态。每个数据流分析可以用五元 $\langle G,L,\sqcup,\sqcap,F\rangle$ 组描述，其中 $G=\langle N,E\rangle$ 表示程序的控制流图，完全格 L 表示安全属性域，F 为流函数集合，$f\in F$ 将一个节点的输入状态映射到该节点的输出状态。在向前(向后)的数据流分析中，$\sqcup(\sqcap)$ 表示 join(split)节点的合并操作。对程序进行数据流分析时，首先设定节点的初始状态，然后沿程序控制流向前(或向后)传播状态信息。当所有节点的状态不再变化时，传播停止，得到程序属性的最终解[8]。

流敏感的数据流分析产生程序属性的 MFP 解(Maximal Fixed Point solution)。以向前分析为例，求解 MFP 解时，首先设定 G 中每个节点的初值(L 的底元素或顶元素)，然后用数据流方程 $State_{In}(n)=\sqcap_{p\in pred(n)}State_{Out}(p)$ 和 $State_{Out}(n)=f_n(State_{In}(n))$ 计算每个节点的输出状态，直到它们不再变化为止。路径敏感的数据流分析产生程序属性的 MOP 解(Meet Over Paths solution)。同样以向前分析为例，首先设定流图起始节点的初始状态，然后沿起流图进行路径敏感的数据流事实传播。假设 $p=[n_0, n_1, \cdots, n_k]$ 表示 G 中一条从节点 n_0 到节点 n_k 的路径，函数 $M: E\rightarrow F$ 映射一个节点到它的流函数，函数 $M(p)=M([n_{k-1},n_k])\circ\cdots\circ M([n_1,n_2])$，那么节点 n_k 的状态 $State(n_k)=\sqcap_{p\in paths(n_0,n_k)}[M(p)](c)$，其中 $paths(n_0, n_k)$ 表示从节点 n_0 到节点 n_k 的路径集合，c 为 n_0 的初始状态。一般情况下，MOP 比 MFP 精确，但 MOP 的计算复杂度是指数级(N 条分支语句可能有 2^N 条路径)，而 MFP 的计算复杂度是多项式级。特别地，当到达某个节点的所有路径的流函数都满足分布律时，即 $f(x\sqcap y)=f(x)\sqcap f(y)$，MFP 解等于 MOP 解。

数据流分析在安全检测中有着广泛的用途。应用数据流分析技术，文[9]检测 C/C++ 程序中的多种安全漏洞，文[10]检测 C 程序中的数组越界漏洞，文[11]对程序安全时态属性进行部分验证。

3.3 约束分析

本质上，类型推断和数据流分析都通过维护变量类型或分析状态之间的约束关系确定程序的属性，因此它们都可以用约束分析方法统一描述。一般情况下，约束分析方法将程序分析过程分为约束产生和约束求解两个阶段，前者利用约束产生规则建立变量类型或分析状态之间的约束系统，后者对这些约束系统进行求解[12]。许多程序分析问题可以用一组局部约束产生规则描述，而约束的求解算法往往独立于任何特定的分析，可以编成标准库函数(如 BANE[13])以利于重用，所以约束分析是一种易于使用的静态分析方法。

约束系统可以分为等式约束、集合约束和混合约束三种

形式。等式约束的约束项之间只存在相等关系，例如 $x=y$。约束系统采用 unification 方法求解，求解时间和约束项的数量基本成线性关系。集合约束把每个程序变量看成一个值集，变量赋值被解释为集合表达式之间的包含关系。一个集合约束系统用 $\wedge_{i=1}^{k} SC_i$ 表示，其中 SC_i 可以是集合表达式的正约束 $e \subseteq e'$ 或负约束 $e \not\subseteq e'$。虽然集合约束的求解时间一般高于 $O(N^3)$，大于等式约束的求解时间，但集合包含关系具有方向性，能够更精确地解释赋值操作的语义。混合约束系统由部分等式约束和部分集合约束组成。

集合约束系统的求解算法都具有相同的策略：对一个初始的约束系统进行反复的变换，直到这个系统变成一个“已求解形式”，即可从这种形式的系统中直接导出解。求解算法的正确性包括 3 个方面：①转换产生的新约束不能改变原约束系统的解；②变换要么完全终止，从而得到已求解形式，要么发现约束不一致而无解；③当不能再进行变换时，所有产生解所需的信息都必须已经包含在最后产生的约束系统中。然而，判断一个任意的集合约束系统是否存在解是一个 NEXPTIME 完全问题[14]。只有限制约束的形式才能保证约束系统一定存在解，并且明显降低算法复杂度。一种最常见并且总存在最小解的约束系统的形式是 $\wedge_{i=1}^{k} L_i \subseteq R_i$，其中集合表达式 $L ::= L \cup L | c(L, \cdots, L) | a | \bot$，$R ::= R \cup R | c(R, \cdots, R) | a | \top$，$a$ 为集合变量，c 为集合构造子，$\bot$ 和 $\top$ 分别表示值域的底元素和顶元素。这种集合约束系统的求解方法是先将约束系统转换为约束图：约束图中每个节点表示约束系统的一个子表达式，每条从节点 L 到节点 R 的有向边表示一个集合约束 $L \subseteq R$。然后利用动态传动闭包算法进行求解，该算法的复杂度为 $O(N^3)$。

文[15]利用集合约束方法检测 C 程序中的缓冲区溢出漏洞。它将字符数组看成抽象数据类型，通过流不敏感、上下文不敏感的约束传播，建立字符数组的空间大小、字符串长度，以及整数取值范围之间的集合约束，并进行定制的快速约束求解算法发现程序中可能发生缓冲区溢出的字符串操作。文[16]对文[15]的方法进行了改进，引入程序切片、别名分析和线性规划求解，提高了检测的速度和精度。

4 各种静态分析方法比较

以上分析了几种主要的静态分析方法，以及它们在安全漏洞检测中的应用。各种程序分析都通过解释程序的抽象语义，建立程序属性的数学模型，再通过求解这个数学模型，确定程序的属性，但它们的属性域、求解方法、分析能力和求解速度等各不相同。

• 类型推断是一种轻型检测方法，只能在程序中传播并检查变量或函数的类型，一般不考虑语句的执行条件和执行顺序，具有最高的分析速度和最弱的分析能力，适合检查属性域有限而且与控制流无关的安全属性，例如安全敏感位置操作是否未经安全检查就引用了不可信数据源的数据等。

• 数据流分析沿程序控制流传播各种变量属性（包括变量类型），这种传播可以是流敏感或路径敏感的，所以数据流分析具有比类型推断更强的分析能力和更低的分析速度，适合检查需要考虑控制流信息而且变量属性之间的操作十分简单的静态分析问题，例如内存访问越界、常数传播等。

• 约束分析具有比数据流分析更强的分析能力，几乎所有的静态分析问题都可以用约束分析方法描述。约束分析方法将分析过程清晰地分成约束传播和约束求解两个部分，前者基于局部的约束规则，后者存在通用的约束求解库，所以约束分析是一种功能强、实现简单的静态检测方法，适合进行模块化、符号化的分析，以及属性之间存在比较复杂操作的静态分析问题。例如，缓冲区溢出检测、安全模型的权限检查等。约束分析的主要缺点是求解算法的复杂度较高，一般在 $O(n^3)$ 以上。

总的说来，约束分析更适合进行软件的安全检测，因为这类应用一般都比较复杂，需要进行整个程序的约束分析。表 1 详细列举了各种分析方法的差异。在实际应用中，几种不同的静态分析方法可以结合起来，共同完成一项检测工作。例如文[5]就同时使用了类型推断和约束分析两种方法：以类型限定词的推断规则作为约束传播规则，进行约束传播，再利用标准的约束求解器发现 C 程序中的格式化字符串漏洞。

表 1 静态分析方法的比较

	抽象域	求解过程	语义约束	检测能力	检测速度
类型推断	类型信息	类型推断	类型规则	弱	很快
数据流分析	数据流事实	数据流事实传播	流函数	强	快
约束分析	变量属性集合	约束的传播和求解	约束规则	很强	慢

静态分析需要在分析效率和分析精度之间进行折衷。进行具体的安全漏洞检测时，不但要选择静态分析方法，还要选择静态分析时使用多少控制流信息。根据分析时考虑的控制流信息的不同，每一种静态分析方法又可分为流不敏感/流敏感/路径敏感和上下文敏感/上下文不敏感等具体类型。

流不敏感的静态分析不考虑语句的执行顺序，只能为每个过程生成唯一的分析结果，因此分析速度最快，但分析精度最低，不能检查依赖于控制流的程序属性，常用于类型推断。流敏感的静态分析考虑语句的执行顺序，为每个程序点生成单独的分析结果，具有比较高的分析精度和较低的分析速度。路径敏感的分析对到达同一个程序点的所有可能路径都生成单独的分析结果，具有最高的分析精度和最低的分析速度。上下文敏感的静态分析为过程的每个调用点生成单独的分析结果。而上下文不敏感的分析不区分过程的调用点，它将同一个过程所有调用点的上下文合并起来，并为这些上下文生成唯一的分析结果。一般来说，分析时考虑更多的程序信息可以提高分析的精度，但同时增加了时空开销。

由于程序的任何非平凡性质都是不可判定的，所以静态分析算法只能给出源程序某些性质的不完全或不精确的解[3]。根据算法对待不确定性质的态度，静态分析算法可以分为保守的和不保守的两种。保守的静态分析算法将所有具有不确定性质的代码标记为安全漏洞，而不保守的静态分析算法将所有具有不确定性质的代码标记为安全的代码。保守分析不会遗漏任何漏洞，但可能导致大量的误报，而不保守分析则相反。目前，多数安全漏洞检测算法是不保守的，以保证报告漏洞的精度。

5 安全语言

相对于其它程序分析方法，静态分析具有较高的分析速度，但在分析复杂的程序属性时，可能存在较多的误报。通过不完全的分析或者更精确的分析可以减少误报，但这些方法又存在漏报多和复杂度高的缺点。于是，有人提出设计更安全语言以提高静态分析的精度，同时不增加检测算法的复杂度。编程语言可以在以下 3 个方面为安全漏洞的静态分析提

供支持:(1)支持程序员编写的注释信息,用以表示程序运行时必须满足的前件条件或后件条件;(2)在静态分析不能确定安全性的位置插入动态检测代码,通过静态和动态联合使用来消除安全隐患;(3)禁止程序使用某些存在歧义的操作,减少静态分析时的不确定性。

文[18]提出一种带安全类型(safe-typed)的扩展 Java 语言,通过静态定型实现信息流安全策略。该语言允许程序员指定数据机密性和数据完整性的使用策略,并由编译器检测程序是否存在违反使用策略的操作。文[19]提出了一种称为 JFlow 的扩展 Java 语言,该语言支持一种分散的多态标签模型。通过运用标签推断规则,JFlow 可以灵活而方便地静态检测程序员给出的信息流注释信息。JFlow 还在程序的二进制代码中插入少量的运行时标签检查来阻止违反标签注释的操作。

文[20]提出一种 C 语言的安全方言 Cyclone,Cyclone 的设计目标是在保留 C 语言的语法和语义的同时,完全消除缓冲区溢出、格式化字符串和内存管理等 C 程序常见的错误。保留 C 语言的语法和语义可以降低 C 程序移植到 Cyclone 程序的开销,并使得 Cyclone 程序继续拥有灵活的数据表示和内存管理手段。为此,该语言需要程序员提供一些程序运行时的状态信息,并在程序的二进制代码中插入少量动态检查代码,以阻止一些复杂应用导致的安全漏洞。即便这样,Cyclone 编译器仍然有可能无法确定某些软件的安全性,从而拒绝编译这些软件。此时,程序员必须在源程序中提供更多的注释信息,以帮助编译器产生安全的代码。

虽然以上的安全语言可以在静态编译时完成大部分安全检查,但仍需要在程序执行代码中插入少量的动态检查代码。文[22]提出了一种可以 100%静态确定内存安全的 Control-C 语言。该语言通过增加一些 C 语言的语义限制,确保指针和动态内存分配的安全,而不依赖程序员提供注释、插入运行时的动态检测代码或者垃圾收集等传统手段。

针对 CC(Common Criteria)安全标准,文[21]提出一种称为 SPARK 的安全语言。该语言是一个 Ada 语言的子集,并且支持多程序注释,但禁止使用运行时库函数。SPARK 的目标是完全消除程序语义中的歧义,使得程序执行的结果和时空开销都是可静态预测的。程序语义的歧义包括浮点运算的结果并不完全按照 IEEE 754 规范产生,还依赖于编译器和软件执行平台;OO 语言的动态绑定技术导致不可静态预测的程序行为等。SPARK 语言已在铁路、航空航天等领域获得了实际应用。

结束语 随着信息技术的不断发展,人们越来越重视计算机系统的安全性,这种需求将持续推动基于静态分析的安全检测技术的发展。当前,人们根据不同的安全漏洞、编程语言、目标软件和应用需求,提出了多种多样的静态检测算法。我们认为,通过不断的比较和改进,人们可能最终提出一种或几种通用的检测算法和工具,这些算法和工具不但能够检测多数类型的安全漏洞,还能够根据程序特点和用户需求,在被测软件的不同程序点动态选择最适合的检测策略,以尽可能获得较好的检测精度和检测速度。

参 考 文 献

1 Mitchell J C. Programming language methods in computer security. ACM POPL, UK, 2001

2 Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. ACM POPL, USA, 1977

3 Rice H G. Classes of Recursively Enumerable Sets and their Decision Problems. Transactions of the American Mathematical Society, 1953(89): 25~29

4 Foster J S, Fahndrich M, Aiken A. A theory of type qualifiers. ACM PLDI, USA, 1999

5 Shankar U, Talwar K, Foster J S, et al. Detecting format string vulnerabilities with type qualifiers. USENIX Security Symposium, USA, 2001

6 Zhang Xiaolan, Edwards Antony, Jaeger T. Using CQUAL for static analysis of authorization hook. USENIX Security Symposium, USA, 2002

7 Johnson R, Wagner D. Finding user/kernel pointer bugs with type inference. USENIX Security Symposium, 2004

8 Aho A V, Sethi R, Ullman J D. Compilers principles, techniques and tools. 编译原理. 李建中, 姜守旭译. 北京:机械工业出版社, 2003

9 Larochelle D. Statically detecting likely buffer overflow vulnerabilities. USENIX Security Symposium, USA, 2001

10 Xie Yichen, Chou Andy, Engler D. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. ESEC/FSE'03, Helsinki, Finland, September 2003

11 Das M, Lerner S, Seigle M. ESP: path-sensitive program verification in polynomial time. ACM PLDI, Germany, 2002

12 Aiken A. Introduction to set constraint-based program analysis. In: Science of Computer Programming, 1999,35(2): 79~111

13 BANE. http://www.cs.berkeley.edu/Research/Aiken, 1998

14 Cormen T H, Leiserson C E, Rivest R L, et al. Introduction To Algorithms, 2001

15 Wagner D, Foster J, Brewer E, et al. A first step towards automated detection of buffer overrun vulnerabilities. Network and Distributed System Security Symposium, USA, 2000

16 Ganapathy V, Jha S, Chandler D, et al. Buffer overrun detection using linear programming and static analysis. ACM Conference on Computer and Communications Security, USA, 2003

17 Musuvathi M, Engler D. Some lessons from using static analysis and software model checking for bug finding. Workshop on Software Model Checking, USA, 2003

18 Zdancewic S A. Programming language for information secutiy:[Ph D]. Cornell University, 2002

19 Myers A C. JFlow: practical mostly-static information flow control. ACM POPL, USA, 2002

20 Jim T, Morrisett G, Grossman D, et al. Cyclone: A Safe Dialect of C. USENIX Annual Technical Conference, Monterey, CA, 2002

21 Chapman R. SPARK-A state-of-the-practice approach to the Common Criteria implementation requirements

22 Dhurjati D, Kowshik S, Adve V, et al. Memorysafety without runtime checks or garbage collection. ACM TECS, USA, 2005