Steelix：基于程序状态的二进制模糊测试

摘要

基于覆盖的模糊测试技术是找到程序中缺陷、漏洞、崩溃的最有效的技术之一。然而，现存的技术受限于覆盖到魔法字节比较（如，字符串相等比较）。一些方法使用重量级的程序分析来突破字节比较的限制，但可扩展性小。这篇论文中，我们提出一种基于程序状态的二进制模糊测试方法，叫Steelix，它增强了测试工具的渗透能力，在可接受的范围内降低了程序的执行速度。实际上，我们用了轻量级的静态分析和二进制插桩来提供覆盖信息、比较过程信息给测试工具。这样的程序状态信息告知测试工具，在输入数据中如何定位魔法字节，如何有效的变异来匹配魔法字节。我们实现了Steelix并在三个测试集上进行评估：LAVA-M测试集，DARPA CGC二进制样本，5个真实程序。结果表明Steelix相比最先进的测试工具有更好的代码覆盖率和漏洞检测能力。此外，我们发现了1个CVE和9个新的漏洞。

CCS 概念

安全和隐私→软件安全工程

关键词

二进制模糊测试，基于覆盖的模糊测试，二进制插桩

介绍

早在1990年代初期[37]，用于商业软件发行前（COTS）找出软件中的缺陷、漏洞、崩溃方面，模糊测试已经变成最有效的，最具扩展性的测试技术之一。它也被广泛的用在主流软件公司，比如谷歌[28]，微软[15]，Adobe[22]，来确保他们软件产品的质量。模糊测试的关键想法是给被测程序喂大量的异常输入数据来触发没预料到的程序行为，比如崩溃或挂起。

现存的模糊测试方法能够通过两个方面分类。第一方面，基于被测程序（PUT）的结构如何被利用，测试工具能被分为白盒测试、黑盒测试和灰盒测试。白盒测试工具（如，[29, 30, 39]）不是访问了PUT的源代码，就是依靠二进制迁移[34]将汇编代码翻译成中间语言。他们通常用重量级的程序分析，如符号执行[43]，来提高有效性，但是却存在扩展性问题。黑盒测试工具（如，[49]）不知道PUT的内部逻辑，但也因此有效性低。灰盒测试工具（如，[8, 24]）介于白盒测试和黑盒测试两者之间。他们用轻量级的程序分析来抽取PUT的部分信息，不会牺牲测试的速度。

另一方面，基于给PUT的测试数据是如何产生的，测试工具能够被分为基于变异的和基于生成的。基于变异的测试工具（如，[8, 23, 29]）以一个预先提供的测试数据集（即，种子）开始，然后通过变异（如，位翻转）集合中的测试数据生成新的测试数据。他们在测试将紧凑且非结构化的数据格式（如，图片）作为输入的程序时有效。基于生成的测试工具（如，[14, 39, 50]）开始没有测试数据，之后基于对输入格式和语法的了解构造测试数据。他们更适合那些高结构化输入的程序（如，XML）。

在这篇论文中，我们注重灰盒的基于变异的模糊测试。最成功的的技术之一就是基于覆盖的模糊测试，它用来轻量级的插桩来抽取程序在执行每个测试数据时的覆盖信息，来决定给哪个可以保留下来继续进行模糊测试。尤其是，如果一个测试数据触发了一个新的基本块的执行，那么它就被认为是感兴趣的并被保留下来；否则丢弃。因此，基于覆盖的测试工具按递增的方式探索PUT的执行路径。AFL[8]是一个先进的基于覆盖的测试工具，并且已经发现了上百的重要的缺陷[16]。

然而，基于覆盖的模糊测试在探索被魔法字节比较保护的路径时受到限制。魔法字节指能够用于比较指令中的输入数据中的字节。比如，字符串”MAZE”在Fig.1a中的程序中被认为是4个魔法字节。这种情况下，AFL必须一次把这四个字节全都变异正确才能够触发崩溃，因为只变异一个，两个，三个，不能够触发新的覆盖。因此，AFL很难触发崩溃。即，AFL需要最多24\*8次变异。困难在于，基于覆盖的测试工具不知道魔法字节在测试数据中的位置并且不知道如何有效的变异魔法字节来匹配成功。

许多先进方法已经将基于覆盖的模糊测试和一些程序分析技术结合来定位难点。比如，Driller[44]用实际执行来解决比较约束。VUzzer[40]用动态污点分析来解决比较约束。AFL-lafintel[17]在LLVM IR 层用程序转换将魔法字节比较转换成多个嵌套的单字节比较。尽管他们都达到了预期结果，但Driller和VUzzer用了重量级的程序分析（即，实际执行会遇到路径爆炸问题，动态污点分析会严重减慢执行速度）；AFL-lafintel在源码层工作并且不知道魔法字节在测试数据的位置。

这篇论文中，我们提出一种基于程序状态的测试工具Steelix1，能够工作在二进制层，能够进入被魔法字节比较保护的路径，只是在可接受范围内降低了执行速度。Steelix的关键思想在于，我们收集了覆盖信息和比较过程信息（即，是否更多的字节正确匹配了魔法字节）。因此，在这篇论文中程序状态代表覆盖信息和比较过程。不论什么时候通过变异一些字节产生了一个测试数据，触发了新的程序状态，我们就推断魔法字节的位置就在被变异的字节周围，并将这个测试数据保留下来。

实际中，Steelix用轻量级的程序分析和二进制插桩将重量级的程序分析取而代之，以此来收集覆盖和比较过程信息作为指导变异过程的反馈。静态分析过滤掉不感兴趣的比较（如，单字节比较），抽取感兴趣的比较信息。基于抽取的信息，二进制插桩来维护PUT中比较操作数的实际值，并在运行时产生比较过程信息。之后测试工具对PUT进行测试，用收集到的反馈来指导相应的变异。

我们通过扩展AFL已经实现了预期的方法，并用两个广为所用的测试集（即，LAVA-M[27]和DARPA CGC 二进制样本[9]）和五个真实程序（即，pngfix，tcpdump，tiffcp，tiff2pdf和gzip）对其进行了评估。在LAVA-M测试集中的4个程序测试中，有3个程序的测试结果Steelix比VUzzer[40]和AFL-lafintel[17]要好，而且平均找出了3倍多的漏洞。此外，在5个真实程序的测试结果中，有3个程序的测试结果Steelix比AFL-dyninst[11]分别多了平均12.7%，9.7%，14.2%的代码覆盖率、函数覆盖率、分支覆盖率。这5个真实程序的魔法字节比较非常普遍。并且在CGC二进制样本和真实程序的测试中，Steelix比AFL-dyninst找到了更多的漏洞。尤其是我们发现了一个CVE和9个新的漏洞，其中有3个没被AFL-dyninst发现。

总的来讲，我们的工作做了如下的贡献。

我们提出了一个基于程序状态的二进制模糊测试方法，来覆盖那些被魔法字节比较保护的路径，仅仅在可接受范围内降低了执行速度。

我们使用了轻量级的静态分析和二进制插桩，来收集覆盖和比较过程信息作为动态回馈，从而指导变异过程。

我们实现并用多个基准和真实程序评估了Steelix，结果与预期相符。

我们在广为所用的真实程序中，发现了1个CVE和9个之前未被发现的漏洞。

概览

在这个部分，我们先介绍一个启发的例子，之后展示我们提出的方法的大体逻辑。

示例

基于覆盖的测试工具，例如AFL，用覆盖信息来决定哪些测试用例应该保留。比如，对于Fig.1b的程序，给测试用例”XXXX”，当第一个字节成功变异的成M的时候，AFL会保留”MXXX”，因为“MXXX”通过了第一个if条件，触发了新的代码覆盖。基于“MXXX”，AFL会生成新的测试用例“MAXX”，这个测试用例也会被保留下来。通过这种递增的方式，AFL能够最终产生测试用例“MAZE”并触发崩溃。

然而，对于Fig.1a中的程序，如果用Fig.1b中产生测试用例的相同方法，AFL会很难触发崩溃，因为正确变异一个字节并不会产生新的代码覆盖。比如，给测试用例“XXXX”，AFL在一些变异之后能够产生“MXXX”，但是“MXXX”并没有触发新的覆盖。因此，AFL会丢弃“MXXX”这个测试用例，尽管已经逐渐接近通过魔法字节比较的条件。这种情况下，AFL必须一下将四个字节都变异正确才能够触发崩溃。AFL最多需要24\*8次操作才能够触发Fig.1a中的崩溃，但是最多需要4\*28次操作就能触发Fig.1b中的崩溃。

为了突破Fig.1a中的魔法字节比较，AFL-lafintel[17]尝试将Fig.1a中的程序转化成Fig.1b中的程序，然后依靠AFL的能力，递增的触发崩溃。然而，如果Fig.1b程序的测试用例包含1000个字节，魔法字节只是其中的四个字节，这种情况在真实程序中很常见，AFL仍然会很难触发崩溃，因为AFL不知道魔法字节在测试用例的位置，无法执行有效的变异。另一方面，Driller[44]和VUzzer[40]分别用实际执行和动态污点分析来突破魔法字节比较。然而，因为实际执行和动态污点分析是重量级技术，Driller和VUzzer能够处理Fig.1a中的程序，但却在真实程序中面临扩展性问题。

通过前面的例子，我们有两个发现。第一，测试用例“MXXX”已经向匹配魔法字节靠近了一步，应该被保留下来，之后继续进行测试。第二，“MXXX”是从“XXXX”变异过来的，表明了魔法字节的一部分在其中第一个字节的位置，其他的魔法字节很有可能在其周围的位置。在这些发现的刺激下，我们提出了一种方法，即收集覆盖和比较过程信息，通过追踪变异时产生新的比较过程和覆盖的字节，推断魔法字节的位置信息，以此来指导变异过程。

方法概览

我们的方法在直接对PUT二进制程序进行模糊测试的过程中，考虑如下两点。第一，PUT的源代码并不是都能获得。通过工作在二进制层，Steelix能够适应于开源和闭源两种程序。第二，在汇编代码中无法得到比较运算数的类型信息。整数、浮点数、字符串/缓冲区变成了字节的比较。因此在汇编代码上做分析比在源代码层面上做分析更明确清晰。

Fig.2给出了Steelix的大致思路，包含了三个部分：静态分析，二进制插桩，模糊测试循环。实际中，静态分析（看3.1）是拿到程序二进制作为输入然后反编译。基于汇编代码，根据一些规则，过滤掉不感兴趣的比较，以便于只有一部分的比较在测试过程中被动态分析。然后，提取感兴趣的比较的信息和基本块的信息，告诉二进制插桩程序在哪儿插桩，插入什么内容。注，基本块被用于收集覆盖信息，插桩方法和目前基于覆盖的模糊测试技术的插桩方式相同（如，AFL）。因此，我们在3.1，3.2不会阐述如何分析和插桩基本块。

静态提取出来的信息，和程序的二进制文件之后就会送去插桩（看3.2）。二进制插桩有两点注意。第一，我们需要用紧凑的方式标记比较过程，比较过程被记录在一块共享内存中，其大小被限制（64KB）以保证效率[8]。第二，我们插桩程序来得到比较运算数的实际值，并且在测试过程中产生比较过程信息。

最终，模糊测试循环（看3.3）得到插桩后的二进制程序，开始测试。特别指出，在执行插桩后的程序之后，测试工具会得到覆盖和比较过程的反馈，根据反馈产生魔法字节的位置信息。覆盖，比较过程和位置信息被用于指导相应的变异过程。即，选择对应的变异算子。

方法论

这部分，我们详细描述Fig.2中的各个部分。

静态分析

静态分析的目的是提供基本块和比较信息，为二进制插桩做准备（3.2）。这里，我们阐明一下比较信息。

比较指令。静态分析先反编译程序的二进制文件。指令集因平台而异。这方面，我们只考虑x86 32位指令集。x86汇编代码中的比较通过cmp/test指令或是函数调用来完成。

test和cmp指令有两个运算数。test指令通过按位与操作并设置标志位的值。比如，test ebx, ebx指令会将0标志位置（ZF）位如果ebx寄存器是0。

cmp指令减去操作数并设置标志位的值。比如，cmp dword ptr [ebp-4], 9 会将0标志位置位，如果ebp-4的内存位置的内容是9。

cmp和test指令的操作数可以是寄存器，内存引用，或是立即数。一个操作数的大小可以是4字节（dword），2字节（word），1字节（byte）。

对于比较字符串和缓冲区的值，程序先将函数参数值入栈，然后调用对应的函数（如，strcmp/strncmp）。

过滤不感兴趣的比较。在Steelix中，我们只对感兴趣的位置插桩，因为插桩减缓程序速度。我们要尽可能的少插桩来减少程序执行的代价。感兴趣的cmp/test指令是那些比较操作数有意义的。下面的规则描述如何过滤不感兴趣的指令。

单字节比较不感兴趣。之前提到过，用于cmp/test指令的操作数可以是1,2,4个字节。单字节比较可以很容易通过默认的位翻转，和算数增/减匹配成功。

函数返回值的比较不插桩。理由是函数中的计算使得比较操作数和输入之间的关系变得不清晰。比如，对于Fig.3a中的程序，in\_num和magic\_number之间的比较是感兴趣的，因为in\_num直接来自输入数据。然而，Fig.3b程序中的比较就是不感兴趣的。这是因为比较用了哈希函数的返回值，而哈希函数和输入数据相关联。为了一个字节在比较中变异正确，可能需要变异输入数据中的多个字节，复杂度并没减少。

抽取比较信息。过滤掉不感兴趣的比较之后，Steelix从剩下的比较中抽取比较信息，通过静态分析扫描汇编代码，产生两个比较信息的列表。

第一个列表保存感兴趣的cmp和test的指令信息。每个列表项有如下形式：

指令地址：运算数1的信息：运算数2的信息

指令地址是该条指令的地址，运算数#的信息存储的是运算数的类型，即，寄存器、内存引用、立即数。它同样保存了一些其他有用的信息用于在运行时能够得到运算数的实际值。比如，在下面的语句中，偏移量-4同样被记录在操作数1的信息中。

cmp dword ptr [ebp-4], 9

第二个列表保存strcmp、strncmp、memcmp的函数调用的信息。每个列表项包含函数调用的地址和函数调用的名字。

函数调用指令的地址：函数名字

注，每个列表项指令的地址告知二进制插桩时在哪儿插桩。

二进制插桩

静态分析提供二进制程序中感兴趣的比较的静态信息，比较运算数的实际值在静态分析之后是未知的，除非运算数是立即数。因此，基于静态提取的比较信息，我们采用插桩来为测试工具提供运行时的反馈（即，得到比较运算数的实际值并产生比较过程信息）。

比较过程。比较过程和覆盖信息被记录在一个共享内存中，其大小被限制（64KB）以保证效率[8]。因为我们为所有比较记录其过程，我们需要用一种紧凑的方式来标记比较过程来适应共享内存。这里，我们定义比较过程为，从比较运算数的第一个或最后一个字节开始算起有多少个连续字节匹配上了。

比如，一个4字节的比较有Fig.4中16个不同的状态。如果将这16个状态都记录下来，代价太大，因为状态的数量会随着魔法字节数量的增大而暴增。为了避免状态爆炸，我们选择性的用一些状态来标记比较的过程。从Fig.4来看，我们能看到这有24条路径从状态1到状态16。根据我们对比较过程的定义，我们用1→2→6→12→16和1→5→11→15→16两条路径上的状态来推断比较过程。

选择这两条路径的原因是，给定任意两个连续的状态，我们能够知道在下一次迭代中应该变异哪个字节。比如，给状态2,（第一个字节匹配）和状态6（前两个字节匹配），我们能够推断下一次变异应该是向前变异，即，第三个字节。相似的，给定状态11（最后两个字节匹配）和状态15（最后三个字节匹配），我们能够推断我们应该向后变异，即，第一个字节。

再有，Fig.4中的相同行的状态能够归并到一个情况。比如，状态2和状态5是单字节成功匹配的情况能够并到一起，不同的是状态2我们应该向前变异，状态5我们应该向后变异，变异的方向能够通过我们上边说的方法来推断。

因此，我们不需要考虑所有的16种状态，我们仅仅需要考虑5种状态来代表4个字节比较的过程：匹配了0,1,2,3,4个字节。这五种状态对应了Fig.4中的第1到5行。匹配0字节，表示没有任何进展。匹配4个字节表示魔法字节被找到。这两种状态之间的情况则是中间步骤。比如，如果我们有在状态2的输入数据，除了状态6之外的其他状态的输入数据都不能看做是比较过程的标志，测试工具会专注的产生一个状态6的输入数据。变异算子具体如何运用过程信息，我们在第3.3部分讨论。通过将比较过程归类成不同的情况，我们将n个字节比较过程占用的共享内存项的数量从2n减小到n+1。

插桩机制

我们给PUT插桩用于为测试产生程序状态反馈。即，覆盖变化和比较过程。在这，我们注重在比较处插桩。

像在3.1.3中介绍的，两个列表的比较信息由静态分析产生。一个列表保存了cmp/test的指令的信息，另一个列表保存了比较函数调用的信息。这两个列表被用于提供插桩指导，在哪插桩，插入什么内容。

特别指出，对于cmp和test指令，我们在他们指令地址之前插桩。插桩内容的逻辑如下。

第一，比较运算数的实际值在运行时被抽取出来。对于一个寄存器运算数，能够通过直接访问寄存器得到其值。对于内存引用运算数，内存地址被计算并且运算数的值通过对内存解引用获得。

之后，运算数的值被用于产生比较过程信息，如3.2.1部分所述。

最终，测试工具通过共享内存被告知比较过程信息。

对于那些通过函数调用完成的比较，因为函数变量很难从栈上获得。我们重写了函数调用，我们的函数和之前的版本有着相同数量和类型的参数。我们的版本和原始版本有着相同的功能，并且加入了一些逻辑来产生比较信息，从而告知测试工具。

插桩能够帮助测试工具保留能够触发程序状态改变的输入数据，允许测试工具一个字节一个字节的匹配魔法字节比较。比如，在一个n个字节的比较中，如果我们想要一次性正确匹配所有的n个字节，搜索空间的复杂度是28\*n。然而如果我们一个字节一个字节的匹配魔法字节，搜索空间复杂度就会减少到n\*28，这是一个可观的减少。因此，插桩能够帮助解决如何变异魔法字节的问题。

模糊测试循环

模糊测试循环得到插桩后的二进制程序并开始测试。插桩内容能够告知测试工具某个输入数据是否更接近比较条件了。这个信息还不足够有效的指导魔法字节的匹配，因为我们不知道魔法字节在输入数据的哪个位置。为了指导魔法字节的位置信息，可行的方法之一就是使用污点分析来抽取比较运算数和输入数据的关联信息。比如，VUzzer[40] 用动态污点分析来收集比较信息。然而，尽管污点分析很强大很精细，但是它不适合快速的程序执行[40]。在此，我们提出了一种方法用插桩内容的反馈来获得位置信息。这个方法用启发式的思路，如果输入数据的一个字节被用于比较过程，那么其他挨着的字节很可能也被用于比较过程。Steelix中，在测试工具做了一次变异之后，如果被告知已经更接近比较条件了，那么就会将新的输入数据和刚刚变异的字节的位置保存下来。当测试工具尝试变异刚刚的那个新输入数据，它会穷尽所有可能来变异它边上的两个字节，即，局部穷尽变异。

算法1给出了测试循环的过程，普通的变异和我们的局部穷尽变异被同时用在算法中。我们用Fig.5作为例子来说明这些变异如何被覆盖和比较过程信息指导。假设用于比较的魔法字符串是“MAZE”并且初始输入中对应的字节是“XXXX”。Fig.5中的圈中的字节是测试工具正在变异的字节。首先，偏移量2位置的字节从“X”变成了“M”，能够很容易的通过普通的变异算子，如位翻转、算数加/减（14行），得到。在这个变异之后，输入数据1产生了。基于覆盖的测试工具不会意识到这种改变，会丢弃输入数据1。然而，Steelix的插桩内容会告诉测试工具魔法字节比较命中1个字节（15行，17行）因此Steelix会保留输入数据1作为一个中间步骤输入数据，保留变异字节的位置，然后丢弃输入数据0（17-21行）。当测试工具再一次把输入数据1从队列中取出的时候，它会得到这个信息：输入数据1是一个中间步骤输入数据，上一次变异位置是偏移量2（5行）。然后，测试工具会在偏移量1的位置尝试所有256种字节变异的可能。然后，测试工具无法得到任何的反馈，因为偏移量1处的字节是一个无关的字节。因此，测试工具会继续变异偏移量3位置的字节，当变异出输入数据2的时候会被告知已经更接近比较条件了。测试工具会保留输入数据2，丢弃输入数据1，然后继续用输入数据2测试（7-13行）。当测试工具再一次把输入数据2从队列中取出来的时候，它会收到两条信息：输入数据2是由局部穷尽变异产生的；输入数据2是通过向前变异得到的（5行）。然后，测试工具会仅尝试偏移量4处字节的所有变异。像输入数据2，输入数据3会产生，然后输入数据3用于产生输入数据4。当输入数据4产生并应用到测试中的时候，会触发一个新的执行路径。基本块插桩内容会告知测试工具，输入数据4使一个新的基本块覆盖了，输入数据4不再是一个中间步骤了。因此，测试工具不再对输入数据4应用局部穷尽变异。

从例子中，我们可以看出有了比较过程信息和位置信息的帮助，测试工具会产生许多输入数据。然而，并不是每个输入是同样重要的。Steelix中，中间步骤输入数据在最后一个魔法字节产生时就没用了。

Steelix一旦通过一个中间步骤输入数据产生了新的数据，新数据更接近比较条件时，就会将原来的中间步骤输入数据从队列中丢弃（12和21行）。

如上所述，位置信息是基于启发式思路的，即认为用于比较的魔法字节在输入数据中是紧紧相邻的。然而，用于一个比较的魔法字节来自输入数据的不同部分这种情况是有可能的。这种情况下，Steelix在使用局部穷尽变异之后不会被告知更加接近比较条件了。Steelix会继续保留中间步骤输入数据，然后应用普通的变异来变异其他字节（14行），并不会把中间步骤输入数据从队列中丢弃。

实现和评估

我们用python、c和c++实现了Steelix。特别指出，静态分析用IDAPython[20]实现，二进制插桩用Dyninst实现。我们在AFL2.33b版本上扩展成为Steelix。所有的实验结果可以通过我们的网站得到[21]。

评估准备

为了评估Steelix的有效性。我们用一些先进的测试工具与Steelix作比较。

评估数据集。我们用了两个广为所用的基准数据集（即，LAVA-M[27]和DARPA CGC二进制样本[9]）和5个真实程序（即，tiff2pdf, tiffcp, pngfix, gzip和tcpdump）进行评估。基准程序里有确切的缺陷，因此能够为工具评估形成一个有价值的语料库。真实的程序用于展示Steelix在处理大程序时的可扩展性和有效性。

LAVA-M数据集。LAVA-M包含了4个有漏洞的Linux工具，即，base64，md5sum，uniq和who。它们通过向已经存在程序的源代码中注入很难达到的漏洞而产生[27]，并且作为评估模糊测试工具的漏洞检测能力的基准。LAVA的作者已经展示基于覆盖的测试工具和基于SAT的方法不能够有效的找到其中的漏洞[27]。最近的测试工具（如，VUzzer[40]）都用的是这个基准。

DARPA CGC 二进制样本。2016年，先进防御研究项目机构（DARPA）[5]举办了网络挑战赛（CGC）。这是第一次所有计算机捕捉到了访问信号。DARPA发行了141个二进制用于质量检测和决赛的17个代表性二进制样本。然而，这些二进制运行在DARPA实验网络研究评估环境下（DECREE），而Steelix依赖Dyninst，很难将Dyninst迁移到DECREE上。尽管TrailofBits团队努力将这些二进制迁移到Linux系统上[18]，可因为迁移问题还是很难设置某些二进制的测试环境。

因此，我们仅用17个代表性的二进制。相比LAVA-M，CGC二进制样本更小并且每个程序包含更多的漏洞，更适合详细分析Steelix是如何检测漏洞的。

真实程序。我们选了5个真实程序，即，pngfix+libpng[3], tcpdump+libpcap[1], tiffcp+libtiff[2], tiff2pdf+libtiff[2]和gzip[4]，基于如下的两个原则。第一，每个程序都是由官方和库文件一同发布的。我们也对库文用于这些程序的库文件进行插桩，来测试库文件中的程序逻辑。第二，程序/库文件代表不同类型的真实程序。pngfix, tiffcp和tcpdump 是关于数据解析程序的，而tiff2pdf和gzip是运算程序。

先进的工具。我们用3个相关的先进工具和Steelix做比较：AFL-dyninst[11], VUzzer[40]和AFL-lafintel[17]。注，Driller[44]和Steelix很像，但它最近发行的[19]只能适应DECREE二进制。因此，我们没和Driller比较。

AFL[8]只能对有源代码的程序进行处理。为了使AFL能够更有效的测试二进制程序，研究者提出了一些扩展，比如AFL-Qemu[12]，AFLPIN[13]和AFL-dyninst[11]。他们之中，AFL-dyninst是最进阶Steelix的因为我们实现方法依赖Dyninst。因此我们和AFL-dyninst在三个测试集上做比较。

VUzzer是最近发布的模糊测试工具，它针对解决魔法字节比较问题。这个工具在我们实验的时候还没发布。因此，我们和VUzzer只在LAVA-M数据集上比较，用他们论文中的数据。

AFL-lafintel，不像dyninst，VUzzer和Steelix可以对二进制程序进行处理，它需要用PUT的源码我们决定和AFL-lafintel比较是因为他们的分割-征服方法和我们工具的比较过程很像。用LAVA-M数据集来给这四个技术之间做比较。

实验基础。我们运行我们的实验在一台8核Intel(R) Xeon(R) CPU E5-1650 v3和8G内存，32位Ubuntu16.04LTS系统上运行。尽管AFL支持主从测试范例，对于所有基于AFL的测试工具，包括Steelix，AFL-dyninst和AFL-lafintel，我们仅用主测试工具示例来进行实验。这不仅和VUzzer看齐，VUzzer目前还不支持并行模糊测试，也是为了减少额外介绍并行模糊测试。

研究问题。实验能够回答下面三个问题。

RQ1.Steelix的漏洞检测能力怎么样？

RQ2.Steelix的代码覆盖怎么样？

RQ3.Steelix在模糊测试循环中的代价如何？

LAVA-M数据集上的测试结果(RQ1)

LAVA[27]的作者用LAVA-M数据集评估了基于覆盖的模糊测试工具（FUZZER）和基于SAT的方法（SES）近乎5个小时，但没有揭露这两个工具的任何引用细节。相似的，VUzzer[40]的作者在LAVA-M数据集上进行了5个小时的评估，但我们在做实验的时候，他们还没将工具发行出来。因此，我们也对AFL-dyninst，AFL-lafintel和Steelix运行5个小时，然后将LAVA，VUzzer论文中的结果照搬过来。

表1显示，对应的技术找到的漏洞个数。第一列表示目标程序的名字。第二列表示表示每个程序总共被注入的漏洞个数。第三到第五列表示LAVA和VUzzer论文中的结果。最后两列给出了AFL-lafintel和Steelix找到漏洞的总数。注AFL-dyninst在5小时内没有找到任何漏洞，表中就没有列出。每个漏洞在LAVA-M中都有唯一的ID，Steelix在实验中找到的漏洞的ID被列在我们的网站上[21]。

从表1中，我们看出在所有的程序测试结果中，Steelix都比SES、AFL-dyninst要好，并且4个程序中有3个程序的实验结果表明Steelix比FUZZER、VUzzer和AFL-lafintel要好。出现这样结果的原因是：

许多注入到执行路径中的漏洞都是被魔法字节比较保护的漏洞。比如，832ID的漏洞通过变异输入数据中偏移量56的字节为“las!”才能够被触发。我们也计算了直接被我们的局部穷尽变异找到的漏洞的数量，结果在显示在表1中最后一列的括号中。我们的局部穷尽变异也能够生成新的路径覆盖但不产生崩溃的输入数据，还能够生成从那些没计算进来的输入数据变异产生的能够使程序崩溃的输入数据。我们可以看到局部穷尽变异发现了46%的漏洞。这标志着在比较过程的指导下，局部穷尽变异算法能够有效的通过魔法字节比较。

相比VUzzer中重量级的污点分析，Steelix由于轻量级的性质，其稍微减慢的执行速度被更强的渗透能力所平衡。更好的是，在5小时的测试中，根据VUzzer论文中的数据，Steelix平均比VUzzer多执行了645倍的输入数据。另一方面，Steelix能够处理cmp/test指令的所有运算数类型（内存引用，寄存器和立即数），而VUzzer智能处理cmp/test指令的立即数运算数类型。实际上，在4个程序中大约80%漏洞涉及立即数的比较，这表示VUzzer错过了20%的比较。

相比AFL-lafintel，Steelix能够知道魔法字节在输入数据中的位置，AFL-lafintel不知道变异输入数据的哪个位置。再有，Steelix过滤掉了不感兴趣的比较并且用紧凑的方式标记比较过程，这对于大型的二进制程序的模糊测试有很大帮助。如，who。AFL-lafintel将多字节比较转换成嵌套的单字节比较并且想PUT中加入了太多基本块。比如，AFL-lafintel在who中插桩99,866处，但是在AFL的共享内存中只有65,536个位置，这阻碍了AFL-lafintel检测PUT的覆盖变化。相比之下，Steelix在who中只插桩了4833处比较和6385处基本块，使用了共享内存中的大约25,000个位置。这解释了为什么Steelix能够在who中比AFL-lafintel找到更多的漏洞，以及我们为什么没有在4.4部分让AFL-lafintel跑大型真实程序。

另一方面，我们可以看到Steelix在uniq上比VUzzer和AFL-lafintel表现差，和FUZZER发现了相同数量的漏洞。相比VUzzer，Steelix中的变异操作对于uniq的输入数据来说太精细了，uniq的输入是一个包含ASCII字符串的文本文件。Steelix中的穷尽变异是在字节层上工作，不能够快速对文件进行大量改变。VUzzer在其遗传算法上使用交叉变异，对输入数据间进行交换字符串块儿操作，使其能够快速覆盖的uniq的代码。相比AFL-lafintel，我们的插桩，通过二进制重写技术，比AFL-lafintel的编译时插桩要慢。尤其是，在5个小时实验中，AFL-lafintel进行了32.6百万次执行，而AFL-dyninst和Steelix分别只有5.3百万次和4.5百万次执行。依靠源代码，速度优势让AFL-lafintel在5小时之内找出uniq内的更多漏洞。

分析LAVA-M的实验结果得知，我们能够正面回答RQ1，Steelix与目前先进的模糊测试工具相比在漏洞检测能力方面表现突出。

在CGC二进制样本上的结果(RQ1)

对于17个CGC代表性二进制样本，我们根据下面原因过滤了一些：

一些二进制程序和其他的重复。比如，CADET00003和CADET00001重复，只有一些次要改变。因此，在这两个二进制程序上的实验结果相同。

一些二进制程序和其他的相互作用。比如LUNGE00005和其他6个二进制程序涉及进程间通信。这样的二进制不适用于模糊测试，所以剔除了。

有些二进制程序很难生成合法的初始化数据，比如一些输入是和部分程序输出相关联的，有些输入是二进制形式。比如，KPRCA00003是一个图像压缩器以可定义的图像作为输入。

过滤之后，我们用8个CGC二进制样本来分析Steelix和现存的基于覆盖的模糊测试工具如何找出漏洞。我们把Steelix和AFL-dyninst在8个程序上跑了3个小时。表2列出了所有的实验结果。特别指出，AFL-dyninst在8个程序中找到了6个程序的漏洞，而Steelix找到了7个程序的bug。

我们详细分析这些二进制文件，提出一些见解。对于YAN0100002，AFL-dyninst和Steelix都没有发现任何漏洞。这个二进制是一个网球运动计算器，这个程序的漏洞是针对“爱国者导弹失败”[7]构建的。只有当一个输入数据能够引起程序进行百万级的浮点数计算时，漏洞才会被触发。因此，漏洞无关魔法字节比较，这种情况下Steelix无法提高漏洞检测能力。

对于KPRCA00001，二进制是一个root64的编码/解码器。漏洞是在root64编码过程中的越界写操作。如果我们提供了一个输入数据能够触发编码操作，AFL-dyninst和Steelix都能在8分钟左右触发这个漏洞。然而，如果初始数据不能够触发编码函数，AFL-dyninst在3个小时内都很难触发漏洞，但是Steelix仍然可以在10分钟左右触发崩溃。因为程序中的page\_root64函数会将输入数据与“encode”和“decode”作比较；Steelix能够捕捉到魔法字节比较，依赖局部穷尽变异来触发崩溃。

从CGC实验结果的分析来看，我们可以正面回答RQ1的问题。给一个好的初始数据，AFL-dyninst和Steelix都能够有效的找到漏洞；然而当初始数据不是能够触发漏洞所期待的时，Steelix因为渗透能力强，更有潜力找到漏洞。

真是层序的结果（RQ1&RQ2）

我们将AFL-dyninst和Steelix在真实程序上比较其找到崩溃、漏洞次数，代码覆盖率。pngfix, tcpdump和gzip测试了24小时，而tiffcp和tiff2pdf测试了72小时因为他们更大。

崩溃和漏洞分析。Steelix分别在tiffcp，gzip，tiff2pdf中找到了500，139和1个不同的崩溃，AFL-dyninst分别在eiffcp和gzip上找到了367和41个不同的崩溃。对于pngfix和tcpdump，Steelix和AFL-dyninst没有触发任何崩溃。AFL中[8]定义,某个崩溃中如果至少一条关联的执行路径没在之前记录的崩溃中出现过，那么这个崩溃被认为是不同的。Fig.7展示了随着时间的推移发现不同崩溃的情况。我们能够看出Steelix始终能够比AFL-dyninst找出更多的不同的崩溃。

在分析这些崩溃，我们发现10个之前未知的漏洞，并且其中之一已经被CVE收录了，这些漏洞和他们的类型列在表3中。特别指出，我们发现2个越界读漏洞，4个越界写漏洞，1个二次释放漏洞，和3个空指针引用漏洞。关于这些漏洞的更多的细节能够在我们的网站[21]看到。然而，AFL-dyninst少找到了3个，这3个都是被魔法字节比较保护的。Steelix就能够在局部穷尽变异的帮助下成功找到他们3个。

覆盖分析。表4报告了AFL-dyninst和Steelix的行覆盖、函数覆盖、分支覆盖。总的来看，Steelix相比AFL-dyninst对于解析程序（pngfix, tcpdump和tiffcp）在这3方面分别提升12.7%，9.7%和11.4%。这比对于计算程序（tiff2pdf和gzip）在3方面的提升要高很多。这是因为解析程序在逻辑上涉及了更多的魔法字节比较，Steelix恰恰就是为解决这个问题而生的。

又一次的特别指出，pngfix的覆盖率的提升是最大的，而gzip的覆盖率提升是最小的。对于pngfix，输入数据是png文件，文件中包含四字节字符串块，其描述用于控制解析过程的数据块的类型。正确的匹配块类型能够提高代码覆盖率。比如，Steelix覆盖了处理zTXt，iCCP和iTXt块的代码，而AFL-dyninst由于缺少一个有效的方式来匹配魔法字节比较，没有产生这些块。

对于gzip，它的主要逻辑是压缩和解压缩数据。不向png文件，zip文件中的局部头字节用于计算而不是比较。这就限制了Steelix相比于AFL-dyninst的提升。不管怎样，gzip是5个程序中最小的，Steelix和AFL-dyninst都能够覆盖gzip中的大部分代码，所以Steelix的提升变得无关紧要。

再啰嗦一下，我们也计算了生成的覆盖到新基本块的新的输入数据的数量和时间的图。注，我们没包含哪些标志中间比较过程的输入数据。结果显示在Fig.8中。我们可以看到因为代价Steelix比AFL-dyninst收敛的更慢（具体细节讨论请看4.5）然而，我们也看到Steelix能够持续产生新的感兴趣的输入，因此在长时间运行中有更多的感兴趣的输入数据。这也是为什么Steelix能够突破魔法字节比较并且渗透能力比AFL-dyninst强的原因。

从真实程序的分析中，我们正面回答RQ1和RQ2的问题。Steelix由于渗透能力强，相比先进的基于覆盖的模糊测试工具来讲，在漏洞检测能力和代码覆盖能力上都强。

代价评估（RQ3）

为了评估Steelix执行速度的代价，我们首先报告Steelix的插桩统计，然后和AFL-dyninst比较输入数据的执行数量。

回想一下，AFL-dyninst在每个基本块插桩，Steelix在基本块和比较插桩。表5报告了基本块的数量（BB列），通过3.1.2两条规则（即，单字节比较（One-Byte CMP列）和函数返回值比较（FRV CMP列））过滤之后的比较的数量，和插桩的比较的数量（Instrumented CMP列）。我们可以看到Steelix多了平均40%的插桩。同时，平均14.2%和3.5%是单字节比较和函数返回值比较，这已经被Steelix过滤了。

我还得再多说一句，Fig.9展示了在每个程序中输入数据执行的数量。我们能看到除了pngfix，Steelix比AFL-dyninst都慢点儿。对于pngfix，Steelix比AFL-dyninst有更多的输入执行。这是因为执行速度同样被输入数据的质量影响。如果测试工具总是执行那些覆盖到执行速度慢的路径的数据时，执行速度会被严重影响。重要的是，多了40%的插桩换来的只是少了11.5%的输入数据执行，但覆盖了更多的代码，找到了更多的漏洞。这表示这个代价是很小的，而且是可以接受的。

从表5和Fig.9分析来看，我们能够正面回答RQ3的问题，Steelix对于执行速度的影响是很小的，而且是可接受的。

讨论

从我们的评估，我们能够总结Steelix在可以接受的代价下相比目前先进的模糊测试工具，具有更好的漏洞检测能力和代码覆盖能力。不管怎样，我们想要特别的强调Steelix不是万能的，不能处理所有的魔法字节比较问题。第一，函数返回值比较，被Steelix过滤了，不能被Steelix处理。第二，启发式的想法不一定总是成立，即输入数据中的魔法字节是仅仅挨着的，因此我们的插桩可能一定程度上对于突破魔法字节比较来说没那么有效了。这种情况下，符号执行或是污点分析能够帮助推断输入数据和程序是如何关联的。因此，Steelix不能在所有情况下取代符号执行或是污点分析的位置，但是可以在一定程度上为他们减负因为不是所有的比较都必须通过符号执行或是污点分析来解决。

相关工作

这个部分，我们关注基于变异的模糊测试，因为Steelix也是基于变异的。注，一系列的技术都是能够提升基于生成的模糊测试[10, 14, 26, 30, 35, 39, 42, 45, 46, 50]的有效性和效率的。

基于符号执行的方法

符号/实际执行是帮助模糊测试的普遍方法。SAGE[31,32]对执行路径用了符号分析并依靠约束求解来生成测试数据。对比可知，一些方法用了符号分析来帮助模糊测试。SYMFUZZ[25]在程序-输入对上应用了符号分析，用来通过输入数据的比特位置来检测依赖。依赖信息用于计算最优变异率。Babić等人用了符号执行来生成输入数据并且直接探索来触发目标的潜在缺陷。

Driller[44]，最接近Steelix的技术之一，结合了模糊测试和实际执行。特别指出，不论何时模糊测试工具被一些魔法字节比较困扰时，它用实际执行来生成一个输入数据来通过比较。不同的是，Steelix尝试在自身的轻量级的插桩内容的指导下通过魔法字节比较。Steelix能够减轻符号/实际执行带来的负担，但是不能取代符号/实际执行的地位因为Steelix不适用某些魔法字节比较的情况。

污点分析方法

污点分析能够提取输入数据和程序逻辑之间的关系。研究人员提出用污点分析来为模糊测试提供指导。BuzzFuzz[29]和VUzzer[40]使用动态污点分析来帮助测试工具定位感兴趣的字节进行变异。同时，一些方法尝试将污点分析和符号执行结合起来共同指导模糊测试。TaintScope[48]是校验和敏感的模糊测试。它定义了用于安全敏感操作的某个字节通过污点追踪，然后再结合具体执行和符号执行生成的输入数据更有可能触发潜在的缺陷。Dowser[33]定位缓冲区溢出和潜流缺陷。它用污点分析来确定影响数组索引的输入字节，之后用符号执行来生成输入数据触发缺陷。相似的，BORG[38]定位缓冲区重写漏洞。它先用污点分析选择缓冲区访问，之后知道符号执行朝着那些访问位置来检测重写。

污点分析能够帮助精准定位魔法字节在输入数据中的位置，但可能会有很高的性能代价并且降低模糊测试工具的执行速度，由于它是重量级技术的属性的原因。不同的，Steelix使用轻量级的插桩来帮助追踪这样的污点信息，虽然不精准但是能够给测试工具提供足够有效的指导。

模糊测试辅助工具

一些辅助技术已经提出了提高基于变异的模糊测试的效率的方法。Rebert等人[41]和Woo等人[49]根据经验提出了怎样的选择算法和模糊测试调度策略能够帮助最大化测试工具的漏洞检测能力。AFLFast[24]加速了AFL通过关注模糊测试过程中的低频路径，允许测试工具在相同的时间内探索更多的路径。Skyfire[47]通过对已经存在的样本的了解来生成更好的种子输入，为那些需要高结构化输入的程序的模糊测试过程做出了贡献。Kargén和Shahmehri[36]用一个不同的角度来进行模糊测试。他们对产生的程序的机械代码进行变异而不是直接在一个合法的输入数据上进行变异。这种方式，他们能够在关于已经编码的输入数据形式的信息指导下，产生高覆盖率的输入数据。Steelix与这些辅助技术不同。我们计划将他们与Steelix结合起来并且评估模糊测试的有效性和效率是否能够提升的更好。

同时，AFL-lafintel[17]在LLVM IR层使用程序转化，将魔法字节比较转化成多个嵌套的单字节比较。这样的转化能够帮助模糊测试工具将已经接近比较条件的输入数据保留下来，这和Steelix很像。然而，这样的转化能够阻止测试工具歧视并丢弃中间步骤输入数据。测试工具会一直保留这种没什么用的中间步骤数据。与之不同，Steelix能够丢弃这种中间步骤数据。同时AFL-lafintel只能操作源代码，Steelix也可以直接处理二进制程序。AFL-lafintel也无法知道魔法字节在输入数据中的那个位置，这限制严重限制了AFL-lafintel的有效性。

总结

这篇论文中，我们提出了一个基于程序状态的二进制模糊测试方法，Steelix。这个程序状态信息不仅包含覆盖信息，也包含比较过程信息。比较过程信息是通过我们的轻量级的静态分析和二进制插桩收集得到的。在比较过程的指导下，相比传统的基于覆盖的模糊测试工具，Steelix能够更有效、更有效率的渗透魔法字节比较，并且将执行速度减缓的代价控制在一个很低的水平。我们实现了Steelix并且通过多种程序评估了Steelix。结果显示，Steelix能够有效的提高模糊测试的漏洞检测能力和代码覆盖能力。

鸣谢

研究受到了新加坡国家研究基金会支持,荣获国家网络安全 R&D 程序(获奖编码No.NRF2014NCR-NCR001-30), 项目编号M4081588.020.500000, 以及中国国家自然科学基金会 (授权码No.61370079) 和NTU Start-Up 组织支持(M4081190).