A Unified Approach to Identifying and Healing Vulnerabilities in x86 Machine Code

Kirill Kononenko
Secure Software Engineering Group
EC SPRIDE / TU Darmstadt
Mornewegstraße 30
64293 Darmstadt
kirill.kononenko@ec-spride.de

ABSTRACT
The security of software systems is threatened by a wide range of attack vectors, such as buffer overflows, insecure information flow, and side channels, which can leak private information, e.g., by monitoring a program's execution time. Even if programmers manage to avoid such vulnerabilities in a program's source code or bytecode, new vulnerabilities can arise as compilers generate machine code from those representations.

We propose a virtual execution environment for x86 machine code that combines information from compositional, static, and dynamic program analysis to identify vulnerabilities and timing channels, and uses code transformations to prevent those from being exploited. To achieve an appropriate level of performance as well as combine analysis results, our approach stores summary information in the form of conditional rules that can be shared among analyses.

Categories and Subject Descriptors
D.4.6 [**Programming Languages**]: Security and Protection – *access controls, invasive software, security kernels, verification.*

General Terms
Design, Performance, Security, Verification.

Keywords
Cryptography, cyber security, cloud computing, timing channels.

1. INTRODUCTION
In modern software systems, vulnerabilities can arise from many sources. They may occur through incorrect or compromised source code or though compromised tools in the programmer's tool chain, such as compilers or runtime libraries. An antivirus program can detect such vulnerabilities only if they match a known signature. If programs generate executable code at runtime, vulnerabilities in such code will be missed. They can be discovered only while the application is running. Another kind of vulnerability relates to timing channels. A timing channel declassifies some part of classified information for another process by modulating its own use of system resources. To solve this kind of problem, we introduce a virtual execution environment that disassembles code and performs a security analysis.

The Vx32 [1] virtual extension environment is an application-level virtual machine implemented as an ordinary user-mode library and designed to run the native x86 code. Applications can link with and use Vx32 in order to create a safe OS-independent execution environment in which to run mistrusted plug-ins or other extensions written in any language that complies with the x86 code. The machine code for Vx32 is compiled with the GNU Compiler Collection for a special architecture, which is based on x86. First, it is parsed and compiled to an intermediate form. This intermediate form represents a sound approximation of the program behavior and can easily be optimized. Although it cannot express the class of computable functions, it is suitable for domain-specific optimizations. This intermediate form represents variants of deterministic behavior of programs. Simplifying this intermediate representation and use of equivalent transformations allows us to optimize for a specific domain. Algorithms for these transformations are described in a special programming language.

MobiCom'12, August 22–26, 2012, Istanbul, Turkey.
ACM 978-1-4503-1159-5/12/08.

2. Approach and Uniqueness
Our analysis is based on the IFDS/IDE framework [2]. Elements in the analysis are theorems in Primal logic [3]. For this fragment of logic we have formulated an efficient algorithm of derivability based on that of Gurevich-Neeman [3]. If information from any classified function propagates towards any function that may declassify this information, this indicates a potential leak of this information. A deterministic pass and a non-deterministic pass of abstract interpretation allow us to perform dynamic verification efficiently in the most commonly used cases.

We represent the task of dynamic binary translation as a problem of analysis of streams of information at run-time in a specific area of code. A deterministic pass optimizes all paths of execution of the program. We then use a non-deterministic pass that represents the program properties as a stochastic process. As a practical instance of this problem, Figure 1 shows an example of accesses to variables that appear in code; on a primary level (compilation time), this requires that the detection of a target as well as the tracking of that target (with a certain level of security and stability) takes place on a secondary level at runtime.

For example, the non-deterministic part of the optimization collects statistics, such as the number of times each block was executed, branch frequencies, and information of past program behavior that can be used for better optimization. These statistics are used for improved value speculation, branch prediction, and static method lookup. Furthermore, the algorithm chooses between concurrent variants of optimization and finds the most beneficial variant. Finally, the goal is achieved in another thread by a computation of the expected results of these concurrent optimizations without distortion of the operating system. Figure 1 shows an example of both a static and a dynamic trace on a lattice

for analysis of live variables. Each element of the lattice has a specific intensity of accesses to variables.

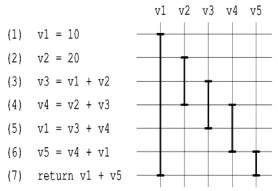

Figure 1 a. Example of a static trace

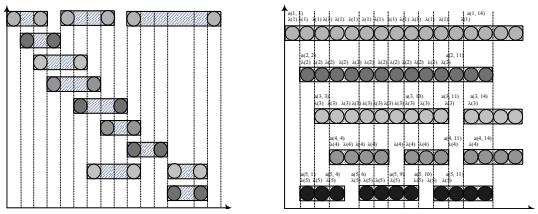

Figure 1 b. Example of a dynamic trace

Figure 2 shows the architecture of an extension of Vx32. The passes are performed in parallel, on different cores, for example. The non-deterministic pass is performed on good test samples, which an expert found very representative, as well as with sampling during execution of the program.

Our extension environment is useful for real-time embedded operating systems and virtual machine runtimes such as the Dalvik and .NET Micro frameworks. In fact, we found that a mid-level intermediate form is not required, as we can represent all properties of the bytecode with executable machine code. This executable machine code can then be parsed and optimized into another version.

There are a few limitations that we consider insignificant. For example, the program is compiled through the use of the GNU Compiler Collection or a compatible compiler. Other significant limitations include the restriction of self-patching and a finite control-flow graph. In addition, any non-deterministic behavior indicates viruses and introduces additional difficulties in debugging of software.

3. RESULTS AND CONTRIBUTIONS

Detection of memory leaks, CHROOT vulnerability, and Trojans is done using the rules of authorization in Primal logic. Detection of stack overflow is done with transformation of the prologue of the function. Additionally, an enhanced Vx32 version makes the code more deterministic from the point of view of the behavior of flags of instructions, accesses of memory, and synchronization of threads.

Instructions that may contain potential timing-channels can be detected statically. For this we use annotations of instructions that classify or declassify information. The weight of a leak is equal to the size of information declassified by the leak.

Timing channels can be fixed with register allocation. Below are the equations used in the optimization of register allocation. The exact solution can be found through the use of a third-party library. Registers are organized in such a way as to minimize the entropy of the timing channel.

$v(i,1)\,x(1) + v(i,2)\,x(2) + \square + v(i,m)\,x(m) \square R(i),\ i \in (1,\dots,N).$

$$min\left(\sum_{j=1}^{m} x(j) f(j)\right).$$

Each equation describes the elements of the lattice from figure 1.

R(i) is the number of available registers for the instruction with an ordinal number *i*, *f(j)* is a random variable, *x(i)* are binary values, *N* is a cardinal number of instructions, and *m* is a cardinal number of variables. Tensor *V* consists of elements *v(i, j)* and is a result of live variable analysis and dead-code elimination.

Vx32 solves these equations with the help of an external utility (a theorem prover). By default we use an approximate solution, which finds a solution in a linear time.

The use of this model for just-in-time compilation yields a significant increase in performance of bytecode execution. For example, use of this model for analysis of MS-CIL and Ruby improves performance of execution of applications in benchmarks.

The results of our research may be useful for the development of just-in-time compilers, interpreters, and compilers with a mixed execution mode of programs for virtual machines such as CLI, Java, Python, Perl, Ruby, LLVM, and Parrot. Further research directions might include the fields of dynamic compilation and dynamic verification. For example, there is the possibility of creating new domain-specific register allocation and code generation algorithms. These algorithms can then be optimized for applications in various domains, for example, in compression and security of binary code.

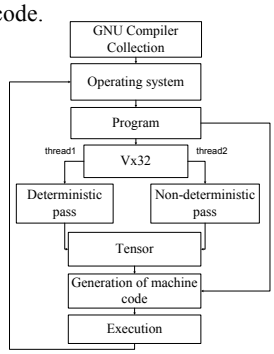

Figure 2. Architecture of extension of Vx32

4. ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) in EC SPRIDE and by the Horst Görtz Foundation and the Hessian LOEWE excellence initiative in CASED.

5. REFERENCES

[1] Vx32 // http://pdos.csail.mit.edu/~baford/vm/.

[2] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167, 1-2 (October 1996), 131-170. DOI=

http://dx.doi.org/10.1016/0304-3975(96)00072-2.

[3] Distributed Knowledge Authorization Language // http://dkal.codeplex.com.