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

Kirill Kononenko
kirill.kononenko@ec-spride.de
Darmstadt Technical University

ABSTRACT

The software systems' security is threatened by a wide range of attacks, such as buffer overflows, insecure information flow, and timing 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 present a unified approach to the secure execution of programs based on a virtual execution environment that combines information from compositional, static and dynamic program analysis in order to identify timing channels, and which uses code transformations to prevent those channels from being exploited. To achieve an appropriate level of performance, as well as combine the analysis results, our approach stores summary information that can be shared between 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 programming systems, vulnerabilities can arise from many sources. The code generated by the compiler can contain multiple vulnerabilities. With the current approach to code optimization, it appears that there are numerous vulnerabilities in applications. They can occur due to an incorrect or faulty source code, or through broken software utilities in a software developer's tools, such as compilers or run-time libraries. A software program may only detect or correct such threats if they have a known static structure.

More difficult vulnerabilities are those that appear dynamically as a result of certain random conditions arising at run-time. Threats of this type require an application of fast and efficient methods of detection and correction. Currently, there are various software packages available [2] (e.g *GenProg*); however, they occupy too much memory and resources. For example, *GenProg* requires infinite mutation spaces. They also do not aim at the correction of bugs and vulnerabilities which arise during the course of program execution. Other methods have a specific character, but they are oriented to find a very limited number of vulnerabilities. If we consider the tasks involved in programming embedded systems, then these methods are not applicable due to the limited amount of resources, memory and CPU time available. In addition, embedded systems are available for limited modifications after their commissioning and fixing any found vulnerability requires a large amount of resources.

To get rid of the limitations of applicable methods, we set a goal of offering a unified approach to identifying and correcting vulnerabilities, which would be more efficient, and which would have a universal character as opposed to individual solutions. It should be possible to formulate this in a unified way and with the means of detecting and counteracting vulnerabilities of various types.

As a starting point, we measured vulnerabilities associated with the program's execution time. A timing channel is a vulnerability that arises as a consequence of a correlation between program's execution's time with the data processed by the program. Such correlation can be applied by an external observer to receive these data by means of analyze of timing parameters of program's execution. An example of the simplest timing channel with leakage of data is shown in Figure 1. Using the public key, by comparing its length with the private key's length, one can find out the private key's length. Private key's length can be found with help of a process similar to a search. Such algorithm sets the public key's parameters and gets the private key's parameters.

The peculiarity of these types of vulnerabilities is that the total time of the program's execution depends on many complex factors: processor data caching techniques; branching operation results prediction; the combination of libraries used; and on the program's execution on concrete hardware platforms. As a result, program execution time dependencies that appear complex from many parameters, lead to timing channels which only appear dynamically.

The easiest way to eliminate a timing channel is to make the time of the execution - the timing channel code - a constant for all variants during the program's execution; then the various states of the program will be externally indistinguishable. However, this approach is not practical in tasks requiring high performance. Practicable, applicable methods of optimizing timing channels result in two problems: detection of the timing channels, and the correction of timing channels in a program at a certain stage of its execution. Detection of timing channels usually happens

```
1 if (privateKey.length == publicKey.length) result = alg1();
2 else result = alg2();
```

Figure 1: Example of source code with timing channel

statically. Algorithms for the correction of timing channels try to counteract them dynamically using the results of their search. Therefore, to eliminate the timing channels we must apply the methods of static and dynamic analyses, which will permit us to work out the direction of the unified approach for identifying and correcting various types of threats.

2. UNIFIED APPROACH

2.1 Overview of Approach

Vulnerabilities' indicators can be static or dynamic found during the execution of the program. For instance, operating system exceptions, failures in hardware, special high-level language constructions, conditional jumps in pre-marked blocks of code. They can be a value of the program's monitors, a large variation in codes execution time, data writing into suspicious areas of code, administrator rights.

Practically applicable methods of vulnerability neutralization need to solve two problems: 1. the detection of a vulnerability and 2. its correction in the program during its execution. The solution should be conducted with much analysis with high computational complexity, which requires huge investment in resources and time which consequently leads to a poorer program performance. Conducting such analysis, particularly interprocedurally, may be performed statically. However, static estimates are not accurate enough and cannot always identify all timing channels.

Therefore, it is possible apply a dynamic estimation, in which optimizations apply information that is unavailable ahead of time before the moment of program execution, by clarifying the results of static analyses and by minimizing additional time delays.

Profiling of code execution time in test examples and dynamic "just-in-time" compilation allows us to conduct an estimation of the execution time of different blocks of code.

Thus we face a problem - combining both static as well as dynamical analyses, where the static phase would be efficiently complemented by the dynamical, in order, to minimize the loss rate of the program and improve the accuracy of identifying threats and eliminate them.

At the core of our proposed approach is the usage of identification and counteraction of the vulnerabilities from a specialized virtual execution environment, that supports a set of compositional, static and dynamical techniques. This execution environment eliminates threats in the code, by combining methods of static detection, and dynamical methods of correction "just-in-time" during program execution.

The execution environment removes threats, starting the system of dynamical code correction, which works in parallel to the main program, providing a constant search for dynamic vulnerabilities.

The virtual execution environment has access to the source code analysis' summaries results, which allows the run-time in a case of faults, creation of new versions of binary codes from the original summaries.

The result of all static and dynamical analyses are stored in a single cache to speed up the re-compilation. Changes in the results of the analysis make it possible to include algorithms of counteraction of threats in newer versions of binary code, or to disable them, in case if it is necessary at some time in the future. The analysis results are stored very compactly and occupy little space, thus we call them summaries of analysis.

Let us demonstrate how the choice of removal method of threats is ensured.

For each type of dynamic vulnerability, as already known and yet unknown, at the development stage of the program are allowed various methods of detection and correction - from complete neutralization, up to a mitigation of vulnerability, and neglect. Controlling capacities of dynamical compilations happens on foundation of usage of combination of special annotations of code blocks, that initialize use of analyses. Annotations are general-purpose. First, annotations turn on, vulnerabilities' profiling, and, analysis of programs work parameters. Second, they designate the boundaries of possible threats in code's blocks (for example, the timing channel boundaries). Thirdly, annotations perform an initialization of threats' monitors. Fourthly, they designate the ways of vulnerabilities' counteraction.

After the vulnerability was indicated and a way of its correction was found, the virtual environment dynamically compiles a new version of the binary code. New program's generation is performed in a linear time due to usage of summaries of analyses.

For example, in the case of malicious code, such as, a call of malicious function, the correction may remove that code entirely, e.g., the call of this function, inserting instead operations of *NOP* instead of the source code. Another instance - register allocation can give a priority for the blocks of code, that create timing channels, or change the order of register allocation. Thus, this approach firstly permits to execute a static version of the program's binary code, in this case triggering the threats' indicator and performing a dynamic compilation of the binary code's next version. As a result, a program is created with removed or mitigated vulnerability.

2.2 Application

As a practical application of vulnerability healing we developed a *Libjit-Linear-Scan* [4, 5] plugin for the CLR compiler. *Libjit-Linear-Scan* supports methods of rapid compilation and analysis based on linear scan.

Firstly, the compiler solved the problem of detecting timing channels and identifying their types. In the second stage, it solved the problem of the dynamic identification of the data input into timing channels, as well as their dynamic neutralization, with a certain level of reliability and stability. Static searches of timing channels occurred interprocedurally, based on a special *IR* and the abstract state machine that it uses.

It generates rules in the language of authorization DKAL [3], which are saved in the summaries for later use during recompilation at run-time. The intermediate representation *Libjit-Linear-Scan IR* permits to conduct dynamic recompilation. Based on line-scanning heuristic rules in the code the environment automatically based on these rules selects the heuristic method for eliminating vulnerabilities, such as the timing channel's neutralization technique during the program execution. For each threat *Libjit-Linear-Scan* uses its unique heuristics. Vulnerability removal rules are inserted in code automatically or manually, by performing heuristical rules.

Vulnerability detection rules and its neutralization are executed automatically with each new pass. Automatic removal of threats consists of a constant heuristic rules verification; including constant search for vulnerabilities and an application of methods to make corrections outside the programs main execution process. For instance, in heuristic correction rule: a data channel's leakage may be due to the fact that its bandwidth is above a predetermined one. Analogically we can do also for other dynamic vulnerabilities.

The sequence of executed steps is shown in Figure 2.

To form a virtual execution environment, we compile the source code in *C* language into *CIL* byte code, which is then dynamically compiled into binary code with the help of the compilation model of *Libjit-Linear-Scan* into intermediate representation *Libjit-Linear-Scan IR*.

A representation of the program in this execution environment allows one to detect threats, which in the virtual environment, are represented as results of analysis, as well as code blocks' sampling values and its properties' statistics. It creates "fat" executable files, that contain executable code, as well as an intermediate representation *Libjit-Linear Scan IR* of the program, suitable for dynamic compilation of optimized versions of the executable code. This allows inserting into the binary code arbitrary changes, as well as produce an instrumentation of code.

For the code in the *Libjit-Linear-Scan IR* representation the virtual execution environment generates summaries with the results of static and dynamic analysis. The intermediate form of *Libjit-Linear-Scan IR* allows one to identify created results of analysis for the original *C* code at any stage of code optimization. After the execution of all static analyses the intermediate form of *Libjit-Linear-Scan* is compiled into the *Vx32* [1] processor's machine code. The *Vx32* architecture enables to execute the binary code more deterministically. Hereby, we obtain on one hand the safety due to usage of the *.NET* execution environment that supports, for example, exceptions handling mechanism. Also, this virtualization environment instead of the original machine code, generates machine code without vulnerabilities, e.g., which does not contain timing channels. The critical part of the code of the original *C* program is being executed in a virtualization mode [1].

2.3 Dynamic Analysis and Tranformations

Static analysis and transformation use algorithms, which have a polynomial complexity, because they are not critical at the time, which is spent on them. On the other hand, a dynamical analysis and transformation must occur quickly and have a low complexity of computation, for example, a linear one. Dynamic elimination of timing channels must use light-weight techniques, but has the advantage of being able to focus only on hazards that actually materialize at runtime. This is done using runtime monitoring and recompilation. Statically prepared information about the potential classification and declassification of data triggers recompilation at runtime. A first recompilation pass instruments the relevant code regions. A subsequent recompilation pass processes the gathered timing data to determine if a timing channel really exists (based on time variability in the gathered profile data), and if needed, eliminates the timing-channel using one of several techniques.

Our dynamic analysis uses a few timing channel techniques for neutralization. First, it can use monitors, which quickly

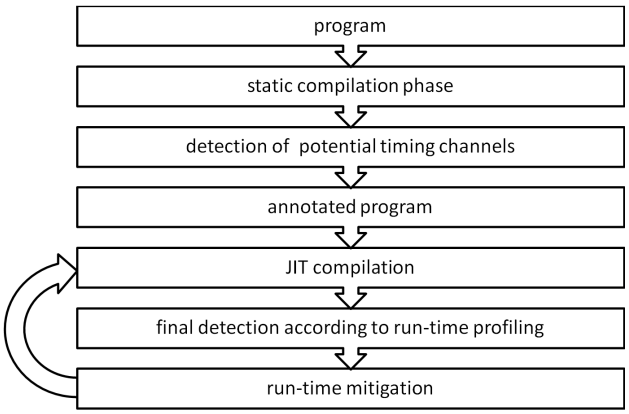

Figure 2: Sequence of passes

analyse emergence of vulnerabilities during execution of the program. Frequently they do not allow one even to use more complex techniques for elimination of timing channels. Secondly, this is a *dynamical compilation* of code inside the timing channel. Dynamical analysis may use techniques of timing channel neutralization using rapid transformation of code in the timing channel, for example, register allocation based on linear scan, that can use statistics. Thirdly, this is *quantification* of the program execution time. If the hardware platform supports creation of parallel processes, then the parallel processes allows a creation of "bundles" of alternative processes. For account of parallel processes we make a dynamical quantification of the time of execution of the code. Fourthly, we can insert into the code random delays and a complex random noise.

3. EVALUATION

To date, we tested the performance of the proposed approach for the detection and neutralization of threats in *C* programs, for which, with help of the software packages *GCC*, *DotGNU* and *.NET Micro Framework*, an environment based on the *CIL* virtual machine has been implemented. For this, we used such means as *Portable.NET*, *Libjit-Linear-Scan*, *NativeProfiler* and *GCC4NET*.

For the generation of test programs with vulnerabilities, we used a special script written in *Ruby* language. It was used for the creation of 10,000 programs, based on code templates for various operations of the virtual machine, such as arithmetic operations, access to arrays, exceptions handling, conditional jumps, loops and function calls, as well as examples of timing channels. We also used the *PNetMark* and *LINPACK* benchmarks, which show the performance of virtual execution environments for various operations of the virtual machine. *PNetMark* is a performance test for various virtual execution environments (CLR). It is based on techniques used in *CaffeineMark* to measure the performance of *Java*. *LINPACK* is a software library for linear algebra operations. It was originally written in *Fortran* and was intended for use on supercomputers in the 1970s and 1980s.

During the testing, we compared the performance of our environment for the programs without and with vulnerabilities.

In the context to timing channels, we measured the program's execution time over varying lengths in our environment. We considered two cases: firstly, this is the case of normal compilation and optimization, without the use of the

analyses' summaries; secondly, this is a case when the analyses' summaries were used. From standard code templates, we created programs of various t lengths: 1, 10, 100, 200, 400, 500, 1000, 2000 and 4000 operations. Then, using our profiler, we measured the program execution time for 100 runs. The program made calculations of the mean and the standard deviation of the program's execution time in these examples. The results of the first series of tests we designate as times $E\,(length)$ and $\sigma\,(length)$. The results of the second test series we designate as times $E^*\,(length)$ and $\sigma^*\,(length)$. The coefficient $k\,(length) = E\,(length)\,/E^*\,(length)$; i.e. the ratio k is equal to the ratio of the program's execution time *without* using summaries of analyses divided by the execution time *with* usage of summaries of analyses. Figure 3 shows the graph of dependence of the values of coefficient k from the program's length.

The graph shows that coefficient k is increasing rapidly. Thus, the analyses' summaries allow one to perform a recompilation of large amounts of code in real time. That time is significant with a large program length. For example, the coefficient k for the *LINPACK* benchmark is more than 20 times. Neutralizing timing channel application techniques show the decrease in the standard deviation of the code execution time in cases of analyses summaries when usage ranges from a few times to 1000 times relative to the code, but which does not take into account the possibility of the appearance of timing channels.

It was confirmed that for all programs where threats were created by us, the environment successfully identified and counteracted all of the proposed vulnerabilities. Herewith, the decrease in test program performance in the vulnerabilities neutralization mode was no more than 10-15%, and certain programs actually executed faster. Experimental data show that our approach increases the security of code execution. It quickly neutralizes vulnerabilities, in case of their identification, spending a few seconds on this, instead of hours and days required for analysis using the vulnerabilities corrections techniques, which perform code compilation, from the beginning without taking in the account the summaries of analyses.

We plan to continue testing our environment to search for other vulnerabilities, for instance: the correction of incorrect generated code by the static compiler *GCC*; the search for program Trojans; the discovery of dynamic viruses; obtaining unauthorized administrative rights; as well as analysing the vulnerabilities related to buffer overflows and accesses of data into unauthorized code areas.

4. CONCLUSION AND FUTURE WORK

For each new vulnerability, we are able to create new rules of search and neutralization of threats that can be checked at each new pass through the code. Vulnerability search rules allows one to create monitors of threats in the binary code. Vulnerability correction rules allows one to introduce changes into the binary code, that counteract the threat.

This system checks the dynamic code patches that performs corrections. In the original program before execution of the program, monitors indicate areas of binary code that can potentially contain vulnerabilities. An example of a vulnerabilities' indication can be a variation of code execution time, an invocation of a banned function, a call of a function with zero address, or access to a null pointer. If the monitors show the threat, then a model's parameter change, which

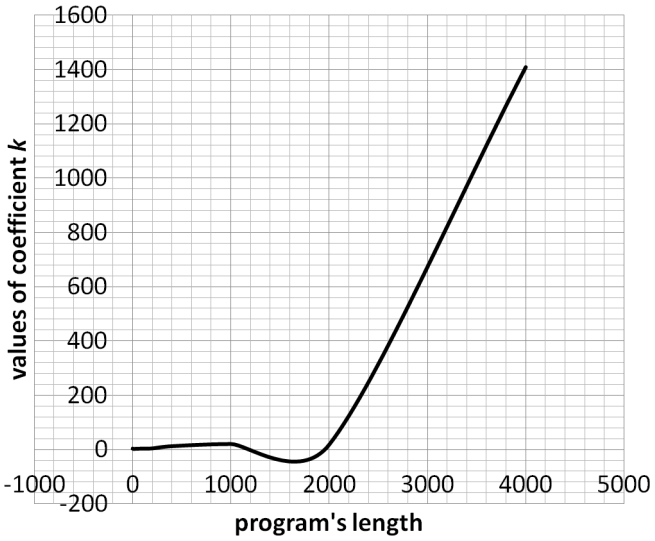

Figure 3: Dependence of coefficient k from the program's length

control the hazard is performed. Then in case of another code execution, instead of the original code, is executed the version of code with the neutralized vulnerability.

As it appears, this approach allows one to find and counteract threats that are caused by incorrect software optimization, due to hardware failures, various types of data leakages, buffers overflows, software viruses and trojans. Basically - these are bugs, which have a dynamic nature and appear only in certain dynamic conditions of program execution, related to inaccuracies in the source or binary code of the program. Alternatively, these are bugs and threats, which can be checked and quickly corrected with an algorithm with linear complexity.

The limitations of the approach is the amount of available memory and the number of available processors that can run in parallel, switching from one task to another on the hardware platform in the mode of application of dynamic patches.

5. REFERENCES

[1] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Commun. ACM*, 55(5):111–119, May 2012.

[2] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[3] Yuri Gurevich and Itay Neeman. Dkal: Distributed-knowledge authorization language. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society.

[4] Kirill Kononenko. Fast compilation in o(n). In *Proceedings of the 2010 International Conference on Theoretical and Mathematical Foundations of Computer Science*, TMFCS '10, pages 51–56, 2010.

[5] Kirill Kononenko. Libjit linear scan: a model for fast and efficient compilation. In *International Review on Modeling and Simulations*, volume 3 N.5, October 2010.