An Empirical Study on the Characteristics of Program Control Flow Data

YONGSUK LEE, GYUNGHO LEE
Department of Computer Science and Engineering
Korea University
145 Anam-ro, Seongbuk-gu, Seoul
SOUTH KOREA
duchi@korea.ac.kr, ghlee@korea.ac.kr

Abstract: Program control flow described in the program dictates its behavior. To have the software behavior dependable and trustworthy, it is critical to secure the program control flow data. Software faults and attacks cause unwanted control flow transfers in program execution via compromised control flow data. However, there are surprisingly little studies on the characteristics of program control flow data. To represent a program control flow, one needs information on the source and the destination, preferably also the path to reach the source, of each control transfer instance. Since they together represent each control transfer instance uniquely, it can be considered a program behavior signature. This paper reports how many unique sources and destinations there are in the execution profiles. The profiled execution traces show that the number of unique control flow transfer instances are surprisingly low, which suggest that confining the program control flow within the set of the unique control flow transfers are feasible in practice. With the control flow confinement, software behavior would be within the expected scope, avoiding unexpected mis-behavior, which leads to more dependable and secure environment for IoT (Internet of Things) and CPS (Cyber Physical System).

Key-Words: Control Flow Graph, Dependability, Indirect Branch, Software Security

1 Introduction

Confining program control flow to ensure that the program execution follows the tested and validated control flow transfers makes a sound principle for developing dependable and trustworthy system. Its premise is that an unexpected control transfer is not allowed to warrant software behavior to be as expected. Considering the emerging popularity of Internet of Things (IoT) and Cyber Physical Systems (CPS), the systems and devices behave as expected in the design has a paramount importance.

Control flow confinement (CFC), ensuring the program execution to follow the reference of a control flow graph (CFG) obtained via profiled execution trace with various input data sets, can make a powerful basis for developing software protection. Unlike the control flow integrity (CFI) that is based on the CFG generated statically [1], CFC is based on only the tested and expected control flows in program execution. The static CFG is bound to be conservative, leaving a room for unintended control transfers included in the CFG, and not able to handle dynamically linked functions properly. Also, the implicit nature of the control flows adds ambiguity to the CFG.

This paper studies the characteristics of the program control flow data that define control flow transfer instances. One particular question is how many unique control flow transfer instances are in the program execution. To represent a program control flow, one needs the source and the destination, preferably also the path to reach the source, of each control flow transfer instance. Since they together represent each control transfer instance uniquely, it can be considered a program behavior signature. If the number of the unique control flow transfer instances is modest in real programs, CFC can make a desirable software protection in practice.

The experimental results reported in this paper are from the complete running of real and full-scale applications under a live operating system. Bochs [2], a full-system Intel Pentium emulator, is used for our profiling study. All the programs are compiled and targeted to dynamically linked x86 binaries, and run under Redhat Linux OS over Bochs. The Linux kernel is modified, so that the hardware emulator becomes aware of process information. All the instructions from the same application image, not just one “representative” process/thread, are tracked. Therefore, more accurate and complete control flow information were collected, even for the multi-threaded applications. The behavior of the dynamically linked library code can be observed as well. Four embedded benchmarks from MiBench [16] have been experimented with.

The rest of this paper is organized as follows. Section 2 describes the background and the motivation of our study. Section 3 explains our choice of the objects for representing control flow transfer instances. Section 4 describes the detection of unexpected control flow transfers with CFC along with its limits and effectiveness. Section 5 presents the conclusion.

2 Background and Motivation

2.1 Control Flow Transfer and Branches

At the machine instruction level, high-level descriptions of control flow transfers are ultimately translated into direct branches and indirect branches for the code binary. The target address of a direct branch is wired in the instruction bits, and points to a single location. The direction of direct branches may be compromised, but the target address cannot be changed. Conversely, an indirect branch reads its target from a memory location or a register. Such target addresses are generated dynamically at runtime. With the contents of the register for indirect branches originated from the memory, an attacker can manage to compromise the control data in memory for the target addresses, by exploiting program's vulnerabilities such as buffer overflow. For example, the target could be replaced with the starting address of a foreign code previously injected or an impossible target address, not following the legitimate execution paths.

Most common indirect branches, in terms of frequency, are the return instructions that read the target addresses saved in the stack. The target of a return is always in the runtime stack in memory, and its location, as well as its value, is known, before the return actually uses it. This makes the return target the most exploited one in software attacks. Many solutions were proposed to protect the return address: from a separate protected copy of the runtime stack, so called "shadow stack", in software [13] or hardware [18], to either guard the return address location [9], or encrypt/hide the return address value [17], [23]. However, fewer works have been undertaken on indirect calls and indirect jumps, called non-return indirect branches in this paper. The major sources of the non-return indirect branches are the uses of function pointers, operations on jump tables in high-level language, non-local jump for library calls, and virtual function mechanisms. An indirect branch, either a return or a non-return indirect branch, provides a desirable point for validating the program control flow.

2.2 Validating Control Flow

The CFGs adopted in the existing control flow validation schemes for software protection have three issues we are concerned about: (1). They are from a static analysis, having rooms for unintended control flows included in the CFG due to the conservative nature of the static analysis; (2). They are for software based control flow transfer validation, incurring a significant performance overhead.; (3). They convey little context information for a particular control transfer instance, allowing the attackers to mount an attack with the legitimate control transfers per the CFG. To alleviate the issues, CFC is based on the CFG in terms of control flow information available from the testing and pilot run of the program during its development. Unlike the static CFG, the CFG generated from the program execution profiles allows that the CFC warrants the tested and expected known program behavior.

To represent a program control flow, one needs the source and the destination, preferably along the path to reach the source. Since they together represent each instance of a control transfer uniquely, it can be considered a program behaviour signature. If the number of the unique control flow transfer instances is modest in real programs, CFC can be feasible in practice.

3 Representing Control Flow

Various control flow related objects, from the branch target address to the complete execution paths or their combination, can represent each control flow transfer instance at indirect branch instruction level. Depending on the scope of the chosen objects, the protection efficacy and overhead can be different. This section considers the objects for more accurate and precise control flow representation but at the same time for little overhead.

3.1 Control Flow Data

A natural object to validate is the target address or the target program counter value (TPC) of each indirect branch instance. Such a validation can prevent the control flow from jumping to the implanted code and/or impossible target address. However, an unexpected control flow transfer might utilize a legitimate target. For example, performing malicious operations via code reuse attacks such as return-to-libc attacks or return oriented programming can be done with legitimate target addresses. TPC alone is not sufficient enough to represent each control flow transfer instance

A more concrete way for representing a control flow transfer is to couple the TPC with its legitimate branch location, i.e., the PC value of the corresponding branch instruction (BPC). As shown in Fig. 1, the pair TPC||BPC can be utilized to detect most control flow compromises including code reuse attacks (CRAs).

Although validating both the branch location and its target address is a popular approach [1], [19], one critical issue is that it samples the program control flow only at isolated program execution points, i.e. at the indirect branches, without considering its context [24]. Consequently, it could miss some elaborate attacks that alter the control flow but still branch from a legitimate indirect branch site to a legitimate target.

To have a context information for a control transfer instance, we include the execution path, besides the indirect branch and its target, into the objects being monitored; only if the pair BPC||TPC of an indirect branch and the execution path that leads to the branch have been validated, is the program allowed to make the control flow transfer. We define the execution path of an indirect branch as the sequence of direction outcomes of the preceding conditional branches to the indirect branch, and denote it as EP (execution path). The set of the PATs, {PAT = (BPC||TPC||EP)}, extracted from the program makes up the CFG. As shown later (in Sec. 4), the inclusion of EP enables CFC to detect non-control data attack.

3.2 Profiling for Control Flow Data

We envision the CFG of the PATs comes as a part of software installation; the software development process generated the CFG of the PATs with various test input data sets. However, without such a provision of providing the CFG in reality, for our work in this paper we have profiled the programs with various input data sets via synthetic input data sets. MiDataSets [14] provides 20 different input data sets that are selected to test most control flow paths in MiBench embedded routines [16]. Four embedded benchmarks from MiBench have been experimented with. To extract the legitimate PATs for the CFG, we had repeated the profiling until the number of PATs of BPC||TPC||EP converges.

We define the number of conditional branch outcomes included in the execution path of an indirect branch as the EP length. A longer EP certainly improves the detection accuracy, and

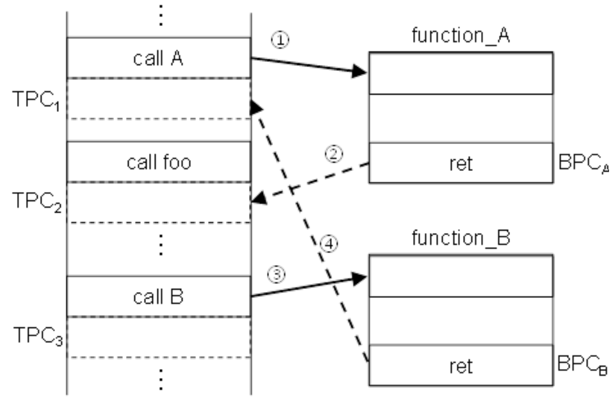

Fig. 1. CFG of the (TPC, BPC) pairs for code reuse attack detection: if a code reuse attack follows the control flow transfer sequence of ①->②->③->④ by compromising the return addresses (② and ④), it will be detected as ② and ④ are not in the CFG.

provides a stronger protection, as long as the branch directions captured in the EP are correlated. However, it comes at the cost of larger storage overhead, as well as slower validation. An excessively long EP may also include unrelated branches, which may provide the opportunity for false positive patterns. Therefore, we must trade the EP length off the overall efficiency. We have profiled the indirect branch's PC and the target PC (BPC||TPC) as completely as possible, and tested the convergence of PATs with various EP lengths. The goal is to have the "truncated" execution path be as short as possible, while still informative enough to reflect the program behavior accurately. Our study suggests that the EP of a short length would be sufficient.

One complication arises in any scheme for generating CFG is how to handle dynamically linked functions. There can be control flow transfers between the executable and the entry address of a function in dynamic libraries, called executable library jumps. Another type of relevant jumps occurs within the library code, called internal jumps. Previous solutions either limit their validation on the static linked functions [1], [19], or track only the internal jumps within the same library, ignoring the executable library jumps [11]. We address this issue of dynamically linked targets, by seeking help from the linker and loader. A target address for the indirect library call could be resolved with only two values. One is the entrance address of the linker, which is always fixed for a given runtime system. The other is the actual address patched by the linker at runtime, which is always fixed in each run. When constructing a PAT of BPC||TPC||EP for an indirect branch for a library call, the TPC can be initialized as the entrance address of the linker for executable library jumps (e.g. PLT0 in PLT). When the linker or loader resolves the address at runtime, it patches both the function pointer table (e.g. GOT) for dynamic linking, and the TPC in the corresponding signature. For the internal jumps, we adopt a similar method adopted in [11], [25] to track the offset, rather than the absolute address, for the TPC. Thus, a later compromise of function pointers related to the dynamically linked libraries can be detected.

Fig. 2 shows the experimental result of profiling of four embedded benchmarks, rawc, dither, toast, dijks, from MiBench [16] and the 20 different input datasets from MiDataSets [14]. It shows the number of PATs and its convergence pace, with respect to the EP length from zero (only BPC||TPC) to 15, i.e. up to 15 conditional branches prior to each indirect branch instance,

The resultant trends have shown that the number of the PATs is modest and limited, and it converges after a reasonable amount of profiling time over different EP lengths. When more than ten branch outcomes are included in EP, the distance between two adjacent curves becomes larger. This probably means that the additional path information is less informative, and is unlikely to be relevant to the indirect branches, as it might add random noise. Moreover, these curves have a greater slope, indicating a slower convergence speed. With the EP length less than 10, the number of PATs, i.e., the number of unique control flow transfer instances, is less than a few thousands, suggesting the control validation of CFC is feasible in practice.

Our experimental results clearly show that the number of PATs does not grow 2^c (c=the number of conditional branches prior to a control flow transfer instance). The growth of the PATs in our experiments were actually sublinear. In theory, the execution path increases 2^c. This suggests that the static CFG has in general a room for including the control flow transfers not intended in the program.

4 Control Flow Confinement

There have been numerous software and hardware proposals to constrain control flow transfers for secure program execution. Most schemes involve identifying, encrypting, and/or tracking the control data [4], [8], [9], [17], [18]. However, it is not always possible to distinguish and track the interested control data accurately, especially for the non-return indirect branches. Non-control-data attacks [7], [21] to compromise the conditional branch decisions to alter the control flow implicitly without compromising the control data. Also, the code reuse attacks can reinterpret the code binaries

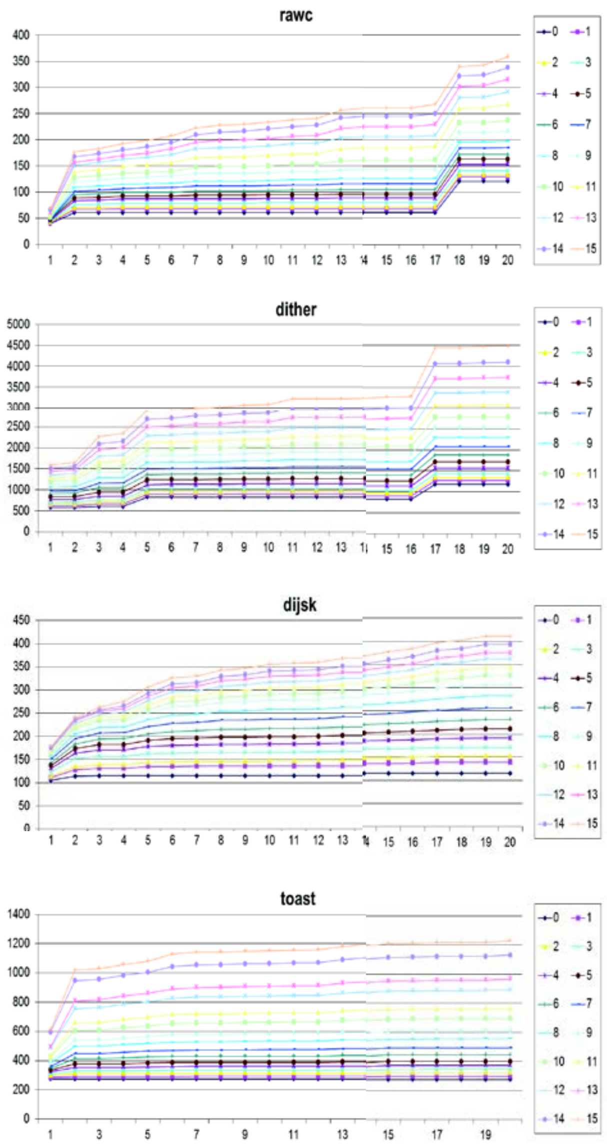

Fig. 2. Convergence of the PATs for embedded applications. It shows the number of the PATs for *rawc*, *dither*, *toast*, and *dijks*, with the 20 different input dataset. Each diagram also shows the results with different EP length from 0 to 15. The vertical axis is for the number of the PATs and the horizontal axis is for the number (in millions) of indirect branch instances encountered.

in memory and rearrange the sequence of the binaries to perform arbitrary functionality [3], [4], [5], [6], [10], [15], [20], [22].

Our CFC scheme per the CFG of the PATs, {PAT=(BPC||TPC||EP)}, is effective against a wide range of control data compromises. First, it is able to detect the control data attacks that introduce a foreign code in the runtime stack or the heap, because the target addresses (TPC) are checked. Checking the branch location (BPC) prevents an adversary from compromising an indirect branch and redirecting the control flow to the existing code

```
SSHD do_authentication()
{   int authenticated = 0;

    while( !authenticated) {
L1: type = packet_read(); //vulnerable

    switch (type) {
       case SSH_CMSG_AUTH_PASSWORD:
L2: if (auth_password(user, passwd))
       authenticated = 1;

       case ..
    }

L3: if (authenticated) break;
    }
    do_authenticated (pw);
}
```

Fig. 3. An example of non-control data attack from a real-world application [7]: A vulnerability in packet_read() at L1 can be exploited, to overwrite the variable of "authenticated" from 0 to 1.

binary as in the code reuse attacks. Including the path information (EP) is a general protection measure to validate the dynamic execution path, based on the correlations among branch instructions, providing context information for a given indirect branch instance. As mentioned in [12], library calls or system calls in many cases are indispensable for an adversary to introduce malicious operations; and a considerable number of realistic run-time systems do invoke library calls through the indirect branches, using a system function pointer table, such as PLT (procedure linkage table) and GOT (global offset table). Thus, checking the execution path before the indirect branches helps to thwart the attacks.

With the EP included for representing each control flow transfer, CFC can detect the non-control data attack compromising the control flow implicitly. Consider the example shown in Fig. 3. It is a non-control data attack to bypass the authentication by compromising the variable "authenticated" that has no direct implication to the control flow [7]: packet_read() at L1 is exploited to overwrite the variable "authenticated" from 0 to 1. So, even with an unauthorized access, i.e., the conditional at L2 is false, the access is granted as authenticated. For the example code, no static CFG can represent the fact that the two conditional branches at L2 and L3 can take the same direction only: if the first conditional at L2 is true then the second conditional at L3 is also true. With the triplet PAT=BPC||TPC||EP (with 2-bit or larger EP), the attack will be detected; a legitimate PAT in the CFG cannot have the EP ending with "01" or "10" for the call do_authenticated. Static CFGs fail in this aspect because they convey no context information regarding a specific control transfer instance. Even a recent context sensitive CFI implementation [24] is not able to handle the non-control data attack

5 Conclusion

Control flow confinement (CFC) ensures that the program execution avoids unexpected control flow transfers, because CFC is based on "dynamic" CFG generated from the execution traces of test input data during program development. Each control transfer instance is defined in terms of the three information pieces, specifically the program counter value for an indirect branch's instance (BPC), its target address (TPC), and the execution path preceding it (EP). The CFG in terms of the program attribute triplets – PAT=(BPC||TPC||EP) is a fine grain context sensitive CFG with no unintended control flow information included.

The number of PATs for the dynamic CFG for CFC is found to be modest, ranging from a few hundreds to several thousands for the embedded programs experimented. Further, our experimental results clearly show that the number of PATs does not grow 2^c (c=the number of conditional branches prior to a control flow transfer instance). In theory, the execution path increases 2^c, and the static CFGs always assume in that way for the sake of conservative flow analysis. This suggests that the static CFG has in general a room for including the control flow transfers not intended in the program. With the trend of more use of IoT and CPS, it is critical to confine software behavior within the known expected behavior space. Our CFC per the dynamic CFG can warrant software behavior within the tested and proven space.

Acknowledgment:
This work was supported in part by the National Research Foundation of Korea (NRF 2015R1A2A2A01). For any correspondence regarding the paper, contact ghlee@korea.ac.kr.

References:

[1] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, "Control-flow integrity principles, implementations, and applications", *ACM Transactions on Information and System Security*, vol. 13, issue 1, Oct. 2009, Article no. 4

[2] Bochs, "The Open Source IA-32 Emulation Project", http://bochs.sourceforge.net/

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” in *Proceedings of the 15th ACM conference on Computer and Communications Security*, Oct. 2008, pp. 27–38.

[4] N. Carlini and D. Wagner, “ROP is still dangerous: Breaking modern defenses”, in *Proceeding of the 23rd USENIX conference on Security Symposium*, 2014, pp. 385-399

[5] N. Carlini, A. Barresi, M. Payer, D. Wagner and T. R. Gross, “Control-flow bending: on the effectiveness of control-flow integrity”, in *Proceedings of the 24nd USENIX conference on Security Symposium*, 2015, pp. 161-176

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return Oriented Programming without Returns”, in *Proceedings of the 17th ACM conference on Computer and Communications Security*, 2010, pp. 559-572.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. “Non-Control-Data Attacks Are Realistic Threats”, in *Proceedings of the 14th conference on USENIX Security Symposium*, Aug. 2005, pp. 12-26.

[8] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, R. Iyer. “Defeating Memory Corruption Attacks via Pointer Taintedness Detection”. in *Proceedings of the International Conference on Dependable Systems and Networks*, June, 2005, pp. 378-387

[9] C. Cowan, C. Pu, D. Maier, J. Walphole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks”, in *Proceedings of the 7th conference on USENIX Security Symposium*, Jan 1998, pp. 5-20.

[10] L. Davi, A. Sadeghi, D. Lehmann, F. Monrose, "Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection", in *Proceedings of the 23rd USENIX conference on Security Symposium*, 2014, pp. 401-416.

[11] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, W. Gong, “Anomaly Detection Using Call Stack Information”, in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May, 2003, pp. 62-75.

[12] S. Forrest, S. Hofmeyr, A. Somayajo, T. Longstaff, “A Sense of Self for Unix Processes”, in *Proceedings of the IEEE Symposium on Security and Privacy,* 1996, pp. 120-128.

[13] M Frantzen and M. Shuey. “Stackghost: Hardware facilitated stack protection”, in *Proceedings of the 10th conference on USENIX Security Symposium*, Aug. 2001, vol. 10, no. 5.

[14] G. Fursin, J. Cavazos, M. O'Boyle and O. Temam, "MiDataSets: creating the conditions for a more realistic evaluation of Iterative optimization", in *Proceeding of the 2nd international conference on High performance embedded architectures and compilers*, 2007, pp. 245-260

[15] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity”, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 575-589.

[16] M . Guthaus, J. S. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite”, in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001, pp. 3-14.

[17] G. Lee and A. Tyagi, “Encoded Program Counter: Self-Protection from Buffer Overflow Attacks”, in *Proceedings of the First International Conference on Internet Computing*, June 2000, pp. 387-394.

[18] Y. Park, Z. Zhang, G. Lee, “Microarchitectural Protection Against Stack-Based Buffer Overflow Attack”, *IEEE Micro*, July 2006, vol 26, no. 4, pp. 62-71.

[19] R. Sekar, M. Bendre, P. Bollineni, D. Dhurjati, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors", in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001, pp. 144-155.

[20] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and Communications security*, Oct. 2007, pp. 552–61.

[21] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347/

[22] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *Proceedings of the 14th International conference on Recent Advances in Intrusion Detection*, 2011, pp. 121–141.

[23] N. Tuck, B. Calder, G. Varghese, “Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow”, in *Proceedings of the 37th annual IEEE/ACM*

International Symposium on Microarchitecture, 2004, pp. 209-220.

[24] V. Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, C. Giuffrida, "Practical Context-Sensitive CFI", in *Proceedings of the 22th ACM conference on Computer and Communications Security*, 2012, pp. 927–940.

[25] D. Wagner, D. Dean, "intrusion detection via Static Analysis", in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001, pp. 156-168.