ISA^{2}R: Improving Software Attack and Analysis Resilience via Compiler-Level Software Diversity

Rafael Fedler, Sebastian Banescu (✉), and Alexander Pretschner

Technische Universität München, Boltzmannstr. 3, 85748
Garching Bei München, Germany
{fedler,banescu,pretschn}@cs.tum.edu

Abstract. The current IT landscape is characterized by *software monoculture*: All installations of one program version are identical. This leads to a huge return of investment for attackers who can develop a single attack once to compromise millions of hosts worldwide. Software diversity has been proposed as an alternative to software monoculture. In this paper we present a collection of diversification transformations called ISA^{2}R, developed for the low-level virtual machine (LLVM). By diversifying the properties crucial to successful exploitation of a vulnerability, we render exploits that work on one installation of a software ineffective against others. Through this we enable developers to add protective measures automatically during compilation. In contrast to similar existing tools, ISA^{2}R provides protection against a wider range of attacks and is applicable to all programming languages supported by LLVM.

Keywords: Software diversity · Software protection · Code obfuscation

1 Introduction

Security can be considered a matter of economics: If the effort or investment required to develop and deploy an attack surpass its promised yields, the attack will not be carried out [11], which holds for all attackers that act rationally in the economical sense. This notion is especially useful as the modern attacker model is increasingly characterized by financial motivation, e.g. monetization of compromised hosts' resources, or indirect economical and political intentions.

Nowadays, malicious adversaries can reverse engineer a program and develop an exploit or patch for it once, then successfully launch it against potentially hundreds of millions of systems, e.g. in the case of widespread vulnerable target software such as browsers, Adobe Reader and Flash Player, or Oracle's Java Runtime Environment, which can be found on almost all private and many commercial computer systems. Thus, by only developing one exploit, millions of systems can be compromised by the attacker. This leads to a big return of investment for many attackers, e.g. exploit developers, software crackers, malware authors and buyers [1]. If the incentive for attack development is lowered, many attacks might not be developed in the first place.

Contributions: In this paper, we present ISA^{2}R, short for *Improving Software Attack and Analysis Resilience*, a collection of diversification transformations for the LLVM compiler infrastructure. ISA^{2}R currently includes 12 transformations diversifying both the process memory layout and executable code of program being compiled. ISA^{2}R generates different program code in each compilation run, which has the same input-output behavior as its input program code. The output programs automatically gain protection against exploits with different aims than availability (e.g., code execution). After transformation, an exploit developed to work on one instance of this program will fail with high likelihood and at most cause an availability issue by making other instances crash. However, the exploit does not achieve its original goal of compromising confidentiality or integrity of the program, e.g. injected code execution. Unlike previous similar work, ISA^{2}R offers protection against a broad range of attacks and supports all programming languages which have an LLVM front-end: C/C++, Java bytecode, C#, Python, Ruby, Objective-C, Go, Swift, Fortran, ActionScript, etc.

This paper is organized as follows: Sect. 2, presents related work. Section 3 describes the transformations offered by ISA^{2}R. Section 4 covers the evaluation of ISA^{2}R. Conclusions and future work ideas are presented in Sect. 5.

2 Related Work

Software diversity as a concept to increase fault tolerance has been an active research area since the 1970s [10]. Software diversity for security purposes, however, has gained momentum only rather recently [2,5]. *Obfuscator-LLVM*[1] is a tool similar to ISA^{2}R. However, *Obfuscator-LLVM*'s 3 transformations aim to protect intellectual property (IP) against *manual reverse engineering*, while ISA^{2}R offers 12 transformations hampering *automatic reverse engineering*, a pre-requisite for developing effective buffer overflow and code patching exploits. Nevertheless, the transformations of these two tools can be merged into one tool since they both operate on LLVM code. *Sandmark*[2] and *Tigress*[3] are two obfuscation and diversification tools. They offer transformations similar to those of ISA^{2}R. However, they are applicable to Java, respectively C source code whereas ISA^{2}R supports all languages with an LLVM front-end. These tools that leverage software diversity concepts in order to increase security have the following drawbacks: Some are limited to specific program languages, while others are limited with regard to attacks they defend against (e.g. reverse engineering, buffer overflow exploits). With ISA^{2}R, we provide one single tool (for all languages supported by LLVM), which offers protection against most attacks aiming to compromise a large user-base of an application by reusing the same attack, and which can be extended to satisfy a wide range of security-enhancing transformation needs.

[1] https://github.com/obfuscator-llvm/obfuscator.
[2] http://sandmark.cs.arizona.edu/.
[3] http://tigress.cs.arizona.edu/.

3 Implemented Transformations

This section presents all ISA^{2}R transformations, which provide protection against each attack category from the taxonomy of Larsen *et al.* [9], i.e.: information leaks, reverse engineering, memory corruption, code injection, code reuse, program tampering. We make ISA^{2}R available to anyone from academia and industry upon request. Most can be controlled by a set of parameters, which we do not describe in detail for the sake of brevity, however, they are described in ISA^{2}R's user manual. One of these parameters is called the transformation probability and specifies the likelihood that a potential target function will be transformed.

Instruction Reordering: By determining the happened-before relation on code and thus the temporal dependencies of instructions, one also determines which statements are independent of each other and can thus be randomized in regards to their execution order. This transformation was created to help against code reuse attacks, as it diversifies the location of instructions relative to the base address of the program similar to Instruction Location Randomization (ILR) [6].

Insertion of Loops with Randomized Iteration Count: This transformation inserts loops into conditional branches (`if/else` statements as well as loop entry/exit conditions). These loops calculate the Collatz series, which, until it converges, greatly varies in the number of iterations until convergence depending on its input character. This transformation is intended to automatically counter timing and, to some extent, power side channel attacks [7,8].

Opaque Predicate Insertion: Opaque predicates are logical predicates whose evaluation is known a priori to the party performing the transformation, but whose evaluation is hard to perform statically [4]. In some cases, as in well implemented pointer arithmetic-based opaque predicates, evaluation may be close to practically infeasible. They can be used to obscure control flow. Often, opaque predicates are used to strengthen other transformations.

Insertion of Bogus Code: This transformation inserts random instructions which do not interfere with the original instructions of a method. It is protected against static analysis using opaque predicates in the condition of `if/else`-statements that either lead to correct instructions or to incorrect ones. This transformation obfuscates the logic of a method, which make reverse engineering harder. However, the added instructions also influence the location of pre-existing program instructions, having a positive effect against code reuse attacks.

Stack Frame Size Diversification: This transformation adds an odd random number of bytes to each stack frame of a binary, being a more fine-grained version of ASLR. This makes buffer overflow exploits applicable to only a fraction of binary instances of a program. Therefore, if the attacker uses the same code injection exploit code that writes a fixed number of bytes to the vulnerable buffer on a diversified program instance, it overwrites the stack pointer (ESP) and the return address (EIP) on the stack with values which will very likely lead to a crash, not achieving the attacker's goal of capturing the control flow of the program. This happens because the offset of ESP and EIP will vary for each diversified binary instance.

Stack Variable Order Randomization: The order of the variables on the stack are randomized, in order to diversify the distance from a vulnerable buffer to important target values such as the copies of ESP and EIP on the stack. This decreases the applicability of a buffer overflow exploit which assumes a fixed variable layout. Additionally, this transformations places all buffers/arrays above all non-buffer variables on the stack, to prevent pointers or variables (used in branch-condition evaluation), from being overwritten if a buffer overflow occurs.

Addition of Bogus Method Arguments: This transformation inserts random arguments in method definitions, removing part of the logical abstractions of the program. The bogus arguments are used by bogus code added to the corresponding method and are finally secured with opaque predicates which make it seem as though the bogus argument actually contributes to the return value.

Method Merging: This transformation breaks the abstraction created by developers (i.e. methods) by merging two unconnected methods into one. The only requirement is that the return type of the two methods is identical. An additional selection parameter is appended to the argument list and all call sites are adjusted accordingly. This makes reverse engineering harder, but also diversifies the location of instructions relative to the base address of the program, breaking certain code reuse exploits.

Shuffling of Method Arguments: This transformation randomly permutes the arguments of a method. It is very useful in combination with the method merging transformation, as it always appends an additional indicator variable to determine which of the contained methods should be executed in the merged version.

Method Cloning: To confuse an adversary, methods are cloned and some – but not all – call sites are adjusted to call the clone. It is advisable to run transformations which diversify the code in order to protect the clone from being identified as a clone of another method.

Randomization of Symbolic Method Names: Symbolic method names provide a reverse engineer with useful information about the functionality of a method. They must sometimes be retained in programs and often so in libraries for linking. This transformation assigns random names to methods and provides a textual mapping to assist in automatic name adjustment.

Proceduralization of Static Data: This transformation removes static buffers in a program, which are never written to. For each of them it creates (1) a key and (2) a keyed generator function. At runtime, before each access to such a buffer, the generator function using the correct key recreates the buffer dynamically. Generator function and key can be protected using other transformations such as bogus code insertion. This transformation protects static data such as passwords, host names, etc. from extraction through static analysis attacks – in order to see such data, the adversary must perform dynamic analysis.

4 Preliminary Evaluation

In order to evaluate the ISA^{2}R collection of transformations described in Sect. 3, we want to answer two questions: (1) Are ISA^{2}R's transformations practical to use, or is the impact on binary file size, runtime performance, or compilation process duration too costly to justify the benefits (if any)? and (2) Are

ISA^{2}R's transformations effective at achieving the formulated goal of diversifying a program such that attacks are not applicable on all instances of that program due to the introduced diversity? We performed a preliminary evaluation to start answering these questions. This includes an efficiency and effectiveness evaluation, detailed in the following. As test compilation targets we used the well-known `ls` utility from the GNU core utilities (version 8.23) and our own implementation of the Sieve of Eratosthenes, as they cover very different code profiles. `ls` was chosen as a non-trivial application (over 3000 lines of C code), which is I/O- and user-interaction centric, with lots of branching and special case treatment. Though lacking a GUI, it is comparable to end-user productivity and web software. As such, we used `ls` to determine compilation time and binary file size overhead. The Sieve of Eratosthenes was selected as our implementation is CPU intensive and heavy on memory operations without any blocking or other interruptions. It represents software with high CPU performance needs and thus was used to measure runtime overhead of ISA^{2}R's transformations.

4.1 Efficiency

Compilation Time: When compared to GCC, the ISA^{2}R transformations each added about 20 to 30 % compilation time overhead on average.Comparing to a pure LLVM build, which is significantly faster than a pure GCC build, the median relative overhead was 16 % higher. The overhead for all single transformations can be found in Table 1. Significant penalties occur when chaining multiple transformations, as the effect stacks. If all transformations are performed, compilation time increases by more than 200 %. We assume, however, that optimizing the implementation of ISA^{2}R can lead to a 50 % reduction of the compilation overhead through caching results of currently redundant calls.

Table 1. Median ($n = 500$) increase in build time, binary size by single transformations

	`ls` build-time relative increase		`ls` binary file size relative increase				
Transformation	GCC	LLVM	O0	O1	O2	O3	Os
AddBogusArguments	30.7 %	48.4 %	0.27 %	0.27 %	0.02 %	0.25 %	0.02 %
CloneFunctions	33.3 %	52.6 %	2.75 %	2.49 %	2.06 %	2.05 %	1.56 %
InsertBogusCode	20.5 %	32.4 %	0.27 %	0.00 %	0.25 %	0.25 %	0.02 %
InsertRandomLoops	81.0 %	127.9 %	7.49 %	3.76 %	4.54 %	4.82 %	2.40 %
MergeMethods	29.7 %	46.8 %	2.00 %	0.60 %	0.30 %	0.03 %	0.91 %
OpaquePredicateGenerator	32.8 %	51.8 %	0.80 %	0.00 %	0.25 %	0.02 %	0.02 %
ReorderInstructions	23.3 %	36.8 %	0.00 %	0.00 %	0.25 %	0.02 %	0.02 %
ScrambleFunctionNames	16.7 %	26.3 %	0.13 %	0.41 %	0.31 %	0.31 %	0.06 %
ShuffleArguments	25.0 %	39.5 %	0.00 %	0.28 %	0.37 %	0.65 %	0.30 %
StackOrderDiversify	17.3 %	27.4 %	0.00 %	0.00 %	0.25 %	0.02 %	0.02 %
StackSizeDiversify	17.2 %	27.1 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
StaticToProceduralData	153.0 %	241.6 %	17.87 %	7.72 %	11.52 %	11.55 %	11.84 %

Binary File Size Overhead: Most transformations did not add significantly to the size of the diversified binary, as can be seen in the right part of Table 1, which presents the median value taken over 500 repeated measurements for each of the follow 5 different optimization levels of LLVM: O0, O1, O2, O3 and Os. Values range from 0 % to a maximum of 17 %, with the average below 5 %.

Runtime Overhead: As stated above, we used an implementation of the Sieve of Eratosthenes which is heavy on memory and CPU processing to measure runtime overhead. The maximum runtime overhead measured was 5 %, except for recursive calls which suffered from a 116 % increase in runtime when the transformation to remove static data and have it recreated dynamically at runtime is applied. All in all, however, runtime penalties where usually below 1 %, except for InsertRandomLoops and StaticToProceduralData, which are affected by recursion in the sieve of Eratosthenes.

4.2 Effectiveness

To demonstrate the effectiveness of ISA^{2}R we chose to design one scenario and corresponding vulnerable program from each attack category in [9]. Figure 1 provides a condensed and simplified version of these programs combined into one. If the user executing this program is the administrator (line 34), then top secret information is printed (lines 35-36). Otherwise, the program takes an input argument which is copied into a statically allocated buffer called `key` (line 32), which is passed as an argument to an ad-hoc string hashing function called `hs()` (lines 14-20), which in turn makes calls to an integer hashing function called `h` (lines 1-12). If the hash of the input argument is equal to a hard-coded constant (line 37) then secret information is displayed to the authenticated user (lines 38-39). To enable a ROP attack [9] the program uses a function that prints a new line in a strange way (lines 21-26). We use this program as a running example for all attack categories presented below, except information leaks described next.

```
 1  unsigned int h(unsigned int v){
 2   uint32_t x = v;
 3   uint32_t y = rotl(v, 8);
 4   uint32_t z = rotl(v, 16);
 5   uint32_t w = rotl(v, 24);
 6
 7   uint32_t t = x ^ (x << 11);
 8   x = y; y = z; z = w;
 9   w ^= (w >> 19) ^ t ^ (t >> 8);
10
11   return w;
12  }
13
14  unsigned int hs(char str[]){
15   int sum = 0;
16   for (int i = 0; i < strlen(str
        ); ++i){
17    sum += h(str[i]);
18   }
19   return sum;
20  }
```

```
21  void strangeWayToPrintNewLine(){
22    const char path[] = "/bin/echo";
23    char *const myArgv[] = {path, NULL};
24    char *const myEnvp[] = {NULL};
25    execve(myArgv[0], myArgv, myEnvp);
26  }
27
28  int main(int argc, char* argv[]){
29    register uid_t uid;
30    char key[20];
31    uid = geteuid();
32    strcpy(key, argv[1]);
33
34    if (uid == ADMIN) {
35      printf("crown jewels");
36      strangeWayToPrintNewLine();
37    } else if (hs(key) == 0xCAFE){
38      printf("here's your cup");
39      strangeWayToPrintNewLine();
40    }
41    return 0;
42  }
```

Fig. 1. Running example illustrating a vulnerable C program

Information Leaks: To assess effectiveness against information leaks, we chose an RSA implementation using a left-to-right square-and-multiply Montgomery multiplication for exponentiation (pseudo-code in Fig. 2) as our transformation target, where m is the plaintext and n is the length of the key in bits. Before transformation, it is vulnerable to a timing side channel attack due to the if-statement inside the for loop, which allows an attacker to extract a private RSA key. We add perturbations to this timing side channel, by adding a loop with a random number of iterations on every conditional branch. The previous dependency of execution time on key bits is distorted. Before transformation, the implementation will exhibit longer timing in 50 % of multiplications, and in 30 % of squaring operations. As multiplication and squaring operations depend on key bits, this is a statistical prediction model for these key bits. After transformation, however, the bit prediction model is altered because of adding instructions (and thus execution time) randomly to either the case where the current key bit is 0, 1, or to branches independent of key bits at all. As long as the attacker has no access to the diversified implementation, it is hard to predict which behavior is induced by which branches. Quantifying the number of bits leaked by a program after transformation is out of the scope of this paper. However, intuitively timing side channel attacks are harder to perform due to this transformation [7].

```
x = m
for i = n - 2 downto 0
  x = x * x
  if (key[i] == 1) then
    x = x * m
endfor
return x
```

Fig. 2. Square and multiply

Reverse Engineering Attacks: Typically imply extracting the key verification algorithm (line 37, Fig. 1) or hard-coded secrets (lines 35, 38, Fig. 1). To protect against reverse engineering, we applied transformations to merge methods, add bogus operands, obscure control flow through opaque predicates, insert bogus code, and replace static data (i.e. hard-coded secrets) through dynamic generator functions. For lack of objective metrics measuring obfuscation strength and to illustrate ISA^{2}R's transformations, we choose to present an example of effects by ISA^{2}R on the program described above. The left part of Fig. 3 shows the control-flow graphs of two functions `h()` (a single basic block at the top) and `hs()` (four basic blocks at the bottom) before transformation. The right part of Fig. 3 shows the resulting control-flow graph after the application of the aforementioned transformations. The gain in complexity is evident: Two disjoint functions are merged into one, and statically an attacker cannot easily determine which basic block will be the one to return the correct value, thanks to the opaque predicates and bogus code. Moreover, the hard-coded values on lines 35 and 38 in Fig. 1 were transformed into encoded strings by the StaticToProceduralData transformation, which eliminates low-hanging-fruit that could be extracted using the Linux `strings` command-line utility.

Memory Corruption Attacks: We constructed a buffer overflow exploit code tailored for the program in Fig. 1. The exploit assumed that the variables on the stack frame of the main function are `key` above `uid` with sizes of 20 bytes, respectively 4 bytes. By giving a 24 byte argument to the program, the call to `strcpy()` on line 32 in Fig. 1 overwrites the value of variable `uid` with the last 4 bytes of the input argument. If these 4 bytes are equal to the value of `ADMIN`

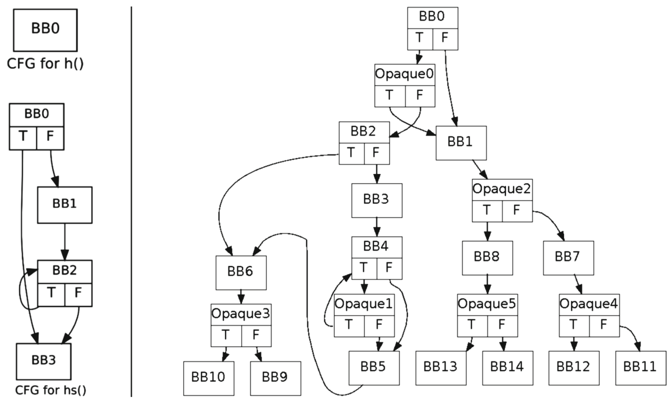

Fig. 3. Control-flow graph of `h()` (top-left), `hs()` (bottom-left) and their merger (right)

the check on line 34 is bypassed and the "crown jewels" are disclosed (line 35). After performing the transformations to randomize stack size and stack variable order of the program from Fig. 1, the same exploit code failed on the diversified instances of the same program, i.e. `uid` would no longer be overwritten because it is placed below `key` and a random size buffer would be placed above `key`. If the size of the inserted buffer was higher than the size of `uid`, the program continued functioning correctly, otherwise a crash occurred. This works reliably for exploit buffers crafted to fit certain byte boundaries. For exploit buffers that consist of repeatedly sprayed return addresses, however, the attacker can still correctly overwrite the EIP on the stack with a probability of 1 divided by the memory address width in bytes. Again, the attacker loses precision in predicting the stack layout which increases the difficulty to construct effective exploits.

Code Injection Attacks: Similar to memory corruption attacks, code injection attacks on the stack fail for the program in Fig. 1. The reason is that due to randomized stack layout and stack frame sizes, it becomes harder for the attacker to predict target address for the injected code. While stack frame size diversification adds a random number of bytes within a certain range to the stack frame, the diversification of the order of variables leads to NOP slides being ineffective, i.e. the execution might try to commence too low in memory, before the buffer holding the NOP slide and the code to be executed. Again, exploits working on the program from Fig. 1 failed in a similar way as for memory corruption attacks, on its diversified instances compiled to allow executable stacks, i.e. the code was placed on the stack but it was not executed successfully.

Code Reuse Attacks: The application from Fig. 1 can be leveraged by ROP exploits, as it calls *libc* functions such as `execve` on line 25. Therefore, we constructed a ROP exploit against this program that bypasses ASLR. By applying transformations to reorder instructions, insert bogus code, insert random loops, and opaque predicates, we randomized code locality as much as possible. Previous gadget addresses were invalidated and the ROP exploit failed with a crash as some gadget addresses now pointed elsewhere, leading to invalid opcodes.

Program Tampering: We crafted two patches for the compiled version of the program in Fig. 1. One rewriting its binary at a specific offset to patch the equality comparison represented by line 37 in Fig. 1, and the other performing pattern matching to locate the correct offset of the call to `hs` inside the binary. By applying the same transformations as in the code reuse attack scenario, we diversified the code section and the fixed offset patch overwrote parts of 2 instructions in the binary program, which failed with a crash when executed. This was not surprising; however, the pattern matching exploit also failed with a crash, as the operand to the `call` opcode it tried to patch was altered through diversification transformations, i.e. `hs` was transformed and its offset displaced.

4.3 Evaluation Summary

As shown in Sect. 4, ISA^{2}R was effective at fending off attacks from each attack category presented by Larsen *et al.* [9] for our example programs. However, this does not mean that ISA^{2}R is effective against all attacks from each category and it may also be ineffective if applied to particular target programs. To further investigate effectiveness we are performing similar case studies using ISA^{2}R on various open source projects. Efficiency measurements showed that most ISA^{2}R transformations have little impact on program file size and execution time which, however, stacks when multiple transformations are applied on the same program. The party who compiles the software bares a significant relative increase since each single transformation takes between 20 % and 30 % more compilation time on average, also resulting in much higher compilation times when multiple (or all) transformations are combined. However, we expect to lower these numbers by around 50 % in the future by eliminating recurring instructions through caching.

5 Conclusion and Future Work

In this work, we presented ISA^{2}R, a collection of software diversification transformations to harden software against attacks. Since ISA^{2}R is based on LLVM it can be applied to programs written in any language supported by LLVM. We have shown through examples that ISA^{2}R is effective at fending off attacks from all software attack categories presented by Larsen *et al.* [9]. Nevertheless, this does not mean that it defends against all attacks.

In our experiments we have constructed attacks which work against one program instance and fail against other diversified instances. Although our evaluation is based on case studies we believe that exploit resilience for the attacks mentioned in our evaluation section generalizes for the majority of programs. It is important to note that vulnerabilities are not removed, but unified exploitation through identical attack vectors is hindered. The more instances exist, the smaller the likelihood of a successful attack for adversaries. This decreases the return of investment for attackers, as one exploit no longer suffices to successfully attack and compromise all installations of a software. This is achieved without prohibitive runtime or file size overhead. However, one significant impact lies in the diversification process, which increases the costs of the software compilation and their distribution to end-users. Usability for software developers is considered high, as they only need to plug in ISA^{2}R into the compilation process.

ISA^{2}R offers 12 transformations for improving resilience against various attacks presented in [9] in a unified way, whereas existing specialized tools support only specific programming languages and are usually commercial or focused on a particular attack. To cover more attacks types, we will add further transformations in the future. ISA^{2}R is available to interested parties upon request.

We intend to add more transformations to protect against further attacks and strengthen existing protections. For example, we plan on integrating *simulation* [3] to further complicate code injection attacks and runtime diversification (also known as *build and execute* [3]) to hamper reverse engineering. More importantly, we plan to assess the resilience of such diversification transformations to adaptable exploits, which are aware of the transformations that can be employed.

We are currently conducting work in the field of *transformation semantics and correctness*. Some transformations presented above can be easily verified to produce valid and semantically equivalent code, e.g. stack size and order diversification, instruction order randomization, etc. Others, however, cannot be trivially verified. As it is of utter importance that diversity transformations do not change functionality or introduce bugs into diversified instances, we are exploring ways to maintain specification semantics and correctness, while changing attack semantics; and to describe formal requirements to ensure these properties.

References

1. Allodi, L., Shim, W., Massacci. F.: Quantitative assessment of risk reduction with cybercrime black market monitoring. IEEE Sec. Priv. Workshops (2013)
2. Banescu, S., Pretschner, A., Battré, D., Cazzulani, S., Shield, R., Thompson, G.: Software-based protection against changeware. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 231–242 (2015)
3. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. **12**(6), 565–584 (1993)
4. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, stealthy opaque constructs. In: 25th ACM SIGPLAN-SIGACT, pp. 184–196 (1998)
5. Forrest, S., Somayaji, A., Ackley, D.: Building diverse computer systems. In: 6th Workshop on Hot Topics in Operating Systems, pp. 67–72, May 1997
6. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.: ILR: where'd my gadgets go? In: IEEE Symposium on Security and Privacy, May 2012
7. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
8. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
9. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: IEEE Symposium on Security & Privacy (2014)
10. Randell, B.: System structure for software fault tolerance. IEEE Trans. Softw. Eng. **1**, 220–232 (1975)
11. Schechter, S.E.: Computer security strength & risk: a quantitative approach. Ph.D. thesis, Harvard University, Cambridge, Massachusetts, May 2004