Obfuscator-LLVM — Software Protection for the Masses

Pascal Junod*, Julien Rinaldini*, Johan Wehrli* and Julie Michielin†
*University of Applied Sciences and Arts Western Switzerland
HES-SO / HEIG-VD / IICT
Yverdon-les-Bains (Switzerland)
{pascal.junod,julien.rinaldini,johan.wehrli}@heig-vd.ch
†Kudelski Security - Nagravision SA
Cheseaux-sur-Lausanne (Switzerland)
julie.michielin@nagra.com

***Abstract*—Software security with respect to reverse-engineering is a challenging discipline that has been researched for several years and which is still active. At the same time, this field is inherently practical, and thus of industrial relevance: indeed, protecting a piece of software against tampering, malicious modifications or reverse-engineering is a very difficult task. In this paper, we present and discuss a software obfuscation prototype tool based on the LLVM compilation suite. Our tool is built as different passes, where some of them have been open-sourced and are freely available, that work on the LLVM Intermediate Representation (IR) code. This approach brings several advantages, including the fact that it is language-agnostic and mostly independent of the target architecture. Our current prototype supports basic instruction substitutions, insertion of bogus control-flow constructs mixed with opaque predicates, control-flow flattening, procedures merging as well as a code tamper-proofing algorithm embedding code and data checksums directly in the control-flow flattening mechanism.**

I. Introduction

Software security with respect to reverse-engineering is a challenging discipline that has been researched for several years and which is still very active. At the same time, this field is inherently practical, and thus of industrial relevance. Protecting a piece of software against tampering, malicious modifications or reverse-engineering is a very difficult task. The main reason is that an adversary has unusual powers, i.e., unlimited access to software, which he can emulate, run step-by-step in a debugger, disassemble, decompile, etc. He can thus read data in memory, including sensitive values such as cryptographic keys, or modify any intermediate values during the software execution and consequently analyze any piece of software with the goal of, e.g., understanding and illegally copying it. One usually refers to such an adversary as a *white-box adversary* [45].

One of the most challenging tasks currently faced by information security researchers consists in designing cryptographic schemes and software techniques that can protect static or running code integrity and confidentiality from white-box adversaries. In the era that sees computations being more and more relocated, a consequence of mobile and cloud computing, solving this challenge would have an enormous impact in practice in terms of the world's overall privacy and security levels. So far, we observe that the problem of protecting the integrity and confidentiality of running code has been attacked from two opposite directions, namely cryptographic and software obfuscation. So-called white-box cryptography techniques have been proposed [10], [45] which allow to instantiate algorithms such as fixed-key implementations from which it is costly to derive the cryptographic key used for the instance (in other words, transforming a symmetric-key cryptographic algorithm in a public-key one). However, most of these proposals have been broken [5], [25], [27], [46]. A particular problem of white-box cryptography is an attack named *code lifting*: when an attacker can execute the implementation of an algorithm under his own control by lifting it from the application, he does not even need to know the original key or understand the algorithm. Fundamental research on cryptographic obfuscation [4], [24], while currently very active thanks to recent breakthroughs [6], [19]–[21], is still in its infancy, and a lot of work has to be done before getting results that can be applied in practice.

Software obfuscation, a maybe more pragmatic approach, does not look for resistance levels comparable to what could be obtained by cryptographic obfuscation (i.e., an unfeasible amount of computations), but merely seeks to increase the reverse-engineering costs in a sufficiently discouraging manner for an adversary.

Popular approaches to secure software consist in relying on trusted hardware [18], [38], [47], such as USB tokens, hardware security modules (HSMs) and smart-cards, or on remote attestation [7], [23], [28], [34], [36]. These solutions have however non-negligible costs and they are not always easy to implement in scenarios such as mobile or cloud computing, or when in non-connected state. In order to secure software running on a mobile phone or on a tablet, where no hardware root of trust is available, one must rely on software-only protection mechanisms which allow to hide data or algorithms and therefore increase the software resistance towards reverse-engineering. Similar considerations apply in cloud computing, where one is running algorithms with potentially sensitive data on computers that are operated by third parties.

Both the academic and industrial worlds have proposed a series of software-only techniques aiming at increasing software security, and we refer the reader to the book of Collberg and Nagra for an extensive review [11]. Those techniques include data encoding, variable splitting and merging, array folding and flattening, conversion of static values to run-time produced values, and mixed-Boolean-arithmetic transforms [12]. Algorithm hiding techniques in the form of a large number of *obfuscating* techniques have been proposed (see [12], [13], [16], among many other works). They all make the reverse-engineering process more costly, but of course not impossible. Some of those obfuscating techniques can even be removed in an automated way [33]. On-demand code decryption (a mechanism also often named "packing") has also been proposed as a method to prevent static analysis of code [3], [22], [37], [44]. These protections are typically broken by applying the static inspection techniques on the decrypted code in memory instead of on the encrypted files on disk. An alternative technique relies on instruction set diversification [2] in which every program version has its instructions encoded using a different byte-code, which is executed using a custom virtual machine. An adversary willing to inspect a program is thus forced to understand the encoding used in a precise instance of the software.

Tamper-resistant software typically uses built-in integrity checks to detect code tampering by guarding the code being executed [8], [26] or by checking that the flow of control through the program is the expected one [9]. However, when these techniques are used to protect standalone applications, their security is rather limited [39], because the expected checksum values and the reaction mechanism is hard-coded in the application itself, where it can be analyzed and altered. Self-checking software can be automatically attacked with hardware support [42] and by attaching a debugger that intervenes in the checksum code and in the decision code [41].

We would like to stress out a new time that, even if none of these techniques will resist in practice to a skilled reverse-engineer, they all significantly increase the reverse-engineering costs, which is probably the best that one can hope at the time of writing.

A. Our Contributions

In this paper, we present Obfuscator-LLVM (`ollvm`), a set of obfuscating code transformations implemented as middle-end passes in the LLVM compilation suite [32]. A majority of the implemented protection techniques are available in the form of an open-source and freely available[1] tool. We have implemented several code transformations aiming at increase software resistance with respect to reverse engineering and tamper-proofing. `ollvm` is currently almost completely language- and platform-independent[2], which means that it can work with all programming languages that are supported by LLVM, notably C, C++, Objective-C and Fortran as well as the `x86`, `x86-64`, `PowerPC`, `PowerPC-64`, `ARM`, `Thumb`, `ARM-64`, `Sparc`, `Alpha` and `MIPS` back-ends, among others. For the moment, `ollvm` implements instruction substitutions, bogus control-flow insertion, basic-block splitting, control-flow flattening, procedures merging and insertion of code tamper-proofing mechanisms, the two last techniques being fully functional but currently not freely available in `ollvm`.

[1]Obfuscator-LLVM is available via the website http://o-llvm.org.

[2]The post-processing operation for the tamper-proofing capabilities, cf. §II-F, is dependent of the target architecture, and thus forms an exception.

B. Related Work

Besides the commercial tools available on the market, there does not exist so many freely available tools that are able to support obfuscation of programs written with traditional programming languages such as C/C++. One prominent recent example is Tigress [15], which is a free, but closed-source, source-to-source virtualizer written in OCaml, supporting the C99 language and implementing many traditional obfuscation techniques. Another example is LOCO [40], a tool based on the Diablo retargetable link-time binary rewriting framework and supporting the `x86` architecture. Working on binaries, LOCO is language-independent and supports control-flow flattening and the insertion of opaque predicates. Sandmark [17] is an open-source, but seemingly unmaintained tool supporting watermarking, tamper-proofing and obfuscation of Java byte-code.

II. Code Transformations

In the following, we quickly review the LLVM compilation suite, discuss code diversification and describe all the obfuscating techniques we have implemented so far in `ollvm`.

A. The LLVM Compilation Suite

The purpose of a compilation suite is to generate code that can be correctly executed on a target CPU [1]. The compiler transforms a high-level code (written in C, C++, Fortran, etc.) into lower-level assembler code, that is then transformed into an object code by the assembler; the resulting object code is then delivered to the linker which is responsible the build the final executable file. In order to perform its duty, a compiler needs to determine if the high-level code has a correct syntax and then, it must correctly generate an assembler code as efficient as possible. The compiler is also required to generate code which complies with the conventions of the target architecture.

LLVM is an open-source compilation suite initially written by Chris Lattner [32] with the goal of providing a modern static-single-assignment (SSA)-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. LLVM is currently supported by a large community of developers and nowadays, it is considered as the main open-source competitor of the GNU compiler collection[3].

LLVM is structured into three main parts: the front-end, the optimizer and the back-end. The front-end is responsible to parse the source code, to verify its correctness and to

[3]http://gcc.gnu.org

build an intermediate representation (IR) of this code that will be delivered to the middle-end, also named optimizer. The two main front-ends supported by LLVM are `clang`, able to compile C/C++ and Objective-C, and front-ends based on the GNU Compiler Collection parsers. The resulting intermediate representation is a kind of universal pseudo-assembler language, named IR code in the LLVM ecosystem, that can be fed to the optimizer and/or to the back-ends. Given some IR code, the optimizer (or middle-end) is then responsible to remove dead or redundant code, inline functions, unroll loops, delete dead loops, simplify the control-flow graph, etc. The result of the optimization process is then delivered to the back-ends, responsible to generate efficient assembler code for the chosen target architecture by taking into account its peculiarities.

Most of the obfuscation and tamper-proofing mechanisms that we have implemented so far in `ollvm` happen in the middle-end. This approach brings several advantages, including the fact that it is language-agnostic and independent of the target architecture. While this way of doing does not allow to implement all possible protections (like, e.g., implementing anti-debugging tricks, that should be done at the code generation level), the advantages to be source- and target-agnostic overwhelm the disadvantages. It goes without saying that more architecture-specific protections can be implemented in the respective back-ends.

Finally, we note that the existence of a back-end able to generate C language instead of assembler code would allow to get rid of the LLVM compilation suite for building executables on architectures for which no back-end is available. Unfortunately, at the time of writing, the C back-end available in the LLVM suite is not supported anymore, but this could change in the future.

B. Bringing Code Diversification

A typical compilation suite is usually deterministic, i.e., compiling two times the same source code results in the same generated machine code. Machine code diversification (see [29], [30] and the references therein) can be brought by several code transformation techniques as soon as they need some randomness. The main benefit is that it is then possible to distribute individual executables images, implementing in some way a weak form of code watermarking. Obviously, the fact that `ollvm` is open-source clearly make the life easier to an adversary willing to derive a new executable that cannot be traced anymore, however, this does not come without some work.

All the code transformations that we have implemented are randomized in some way. We have chosen to implement a simple cryptographically secure pseudo-random generator (PRNG) consisting of the AES-128 [35] block cipher operated in counter mode. The seed is defined to be the AES key value, and the PRNG is either seeded thanks to the underlying operating system (through `/dev/urandom` or the Windows CryptoAPI, respectively), or with a user-provided seed value transmitted through a compilation flag. Hence, providing the same seed between different compilation runs allows to keep the process deterministic.

TABLE I: Instruction substitutions implemented in `ollvm` (pseudo-C notation). `a`, `b` and `c` are integer variables, while `r` is a pseudo-random value.

Operator	Equivalent Instruction Sequence

`a = b + c`	`a = b - (-c)`
	`a = -(-b+ (-c))`
	`a = b + r; a += c; a -= r`
	`a = b - r; a += c; a += r`

`a = b - c`	`a = b + (-c)`
	`a = b + r; a -= c; a -= r`
	`a = b - r; a -= c; a += r`

`a = b & c`	`a = (b^ !c) & b`
`a = b | c`	`a = (b&c) | (b^ c)`
`a = b ^ c`	`a = (!b&c) | (b&!c)`

C. Instructions Substitution

Instructions substitution forms maybe the simplest obfuscation technique that can be imagined. It consists in replacing standard binary operators, like arithmetic or Boolean ones, by functionally equivalent but more complicated sequences of instructions. Instruction substitution is available through the `-mllvm -sub` option in `ollvm` and currently supports integer additions and subtractions as well as the Boolean operators AND (&), OR (|) and XOR (^). The density of substitutions can be parametered, which allows to avoid a too large impact on the resulting machine code size and performances. Substitution of floating-point operators is not supported, as it brings additional numerical inaccuracy, i.e., the resulting code can no more be considered as functionally equivalent.

This kind of obfuscation is rather straightforward and does not add a lot of resistance to reverse-engineering, as it can easily be circumvented by re-optimizing the generated code. This explains why our instruction substitution pass is run after all LLVM optimization passes. Nevertheless, given the fact that it is possible to generate several equivalent expressions for a given operator (see Table I for a list of the currently implemented ones), choosing one at random is an easy way to bring code diversification in the resulting code machine. Furthermore, an additional benefit is that it can render more difficult the task of automated searching specific machine instruction patterns that are commonly used in symmetrical ciphers, like XORs.

D. Bogus Control Flow Insertion

Essentially, we have implemented the algorithm described in [11, §4.3.4]. Bogus control flow insertion consists in modifying the control flow graph of a function by adding an conditional jump construct that either points either into the original basic block or to a fake basic block looping back to the conditional jump block. An opaque predicate [14], i.e., an expression evaluating always to the same value but hopefully difficult to statically reverse-engineer, is responsible to ensure that at run-time, only the original basic block is executed. The

```c
#include <stdlib.h>

void f(int x){
    int i;

    for(i=0; i < x; i++) { printf("\%d",i); }
}

void g(int x){
    printf("\%d", x);
}

int main(int argc, char** argv) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    if((a^b)+12 == 14) { f(a|b); }
    else { g(a&b); }

    return 0;
}
```

Fig. 1: Toy example in C

opaque predicate also ensures that the optimizer is unable to simplify the resulting call graph by identifying the dead code.

This transformation can be accessed with the `-mllvm -bcf` compilation flag, and it can be parametered in various ways, including the density of insertions, the number of iterations, etc. The effect of inserting bogus control flow is best seen when considering the simple C code source in Fig. 1. The control flow graph of the function `f()` before and after modification is illustrated in Fig. 2. One can note that the control flow graph has been considerably complicated, although most of the added code is dead and will never be executed.

E. Control Flow Flattening and Basic-Block Splitting

The idea behind code flattening is to break the control flow graph of a function by removing all easily identifiable conditional and looping structures [31], [43]. Basically, this is achieved through the use of a large `switch()` construct responsible to route the code control flow through the proper basic blocks depending on a routing variable. At the end of each basic block, the routing variable used in the dispatcher is set in a way that the flow will jump to the correct next basic block. As an illustration, Fig. 3 represents a flattened version of the `f()` function of our toy code. Control-flow flattening functionalities are available through the `-mllvm -fla` compilation flag.

Basic-block splitting capabilities, available through the `-mllvm -splitNum` compilation flag, are also implemented, which can be used to further break the code structure by artificially increasing the number of basic blocks in a function.

Note that, in practice, the resistance to automatic reconstruction of flattened code is very sensitive to the way how the routing variable is updated. Hard-coded static values offer almost no resistance, while updating the routing variable with dynamic values, such as the results of a `check()` routine (see §II-F) used in the tamper-proofing mechanism will force a reverse-engineer to perform a dynamic analysis of the executable.

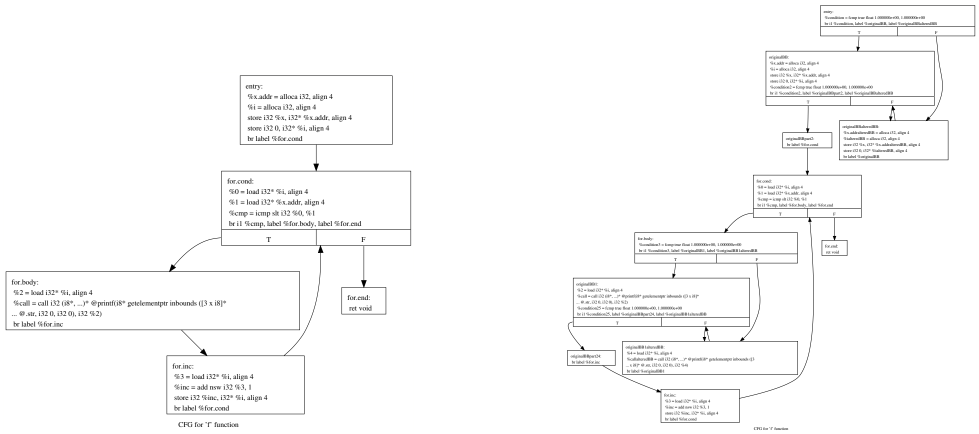
F. Code Tamper-Proofing

The goal of code tamper-proofing techniques consists in ensuring at run-time that machine code has not been modified. This behaviour is implemented through two mechanisms. The first one, `check()`, is responsible to detect a modification, while the second one, `respond()`, has to take an action when the code integrity is detected to be hurt. In order to make more difficult reverse-engineering operations, the routines `check()`, `respond()` and the real response (kill the program, tamper its result, etc.) should be temporally spaced. The ideal situation is to have many `check()` and `respond()` routines spread throughout the code, and possibly in a way where they are inter-dependent.

In `ollvm`, we have implemented a tamper-proofing mechanism that is integrated with the code flattening capabilities. It means that various `check()` routines are inserted into the code, and their result are used to dynamically update the variable responsible to route the control flow in a previously flattened code. Currently, each basic block needs to have at least one `check()` call, but we plan to remove this constraint in the future, for obvious performance reasons.

Currently, a `check()` routine is implemented as the computation of a 32-bit CRC checksum over a segment of machine code, where the begin and the end of the segment are chosen at random, respecting a predefined maximal segment length. Furthermore, the location of the `check()` routine is randomly chosen within the basic block. On the Intel `x86` 32-bit and 64-bit architectures, we rely on the `CRC32` dedicated instruction, which allows to significantly improve the performances of a check. Note that the GF(2)-linearity of CRCs allows us to implement the check of inter-dependent code segments (i.e., segments containing a `check()` routine can also be checked).

The result of a `check()` is used to update the basic block routing variable. It means that the result is computed at run-time; if a basic block contains more than one `check()`, the results are combined with an XOR operation. To improve the protection, the routing variable is not only computed dynamically but also depends on a static value generated at compilation time. More precisely, that static value s is computed as $s = \rho \oplus d_1 \oplus d_2 \oplus ... \oplus d_n$, where ρ is the value leading to the next basic block, and d_i, with $1 \leq i \leq n$, are the 32-bit results of the n `check()` routines present in the current basic block. Note that the values $d_1 \dots d_n$ are known only after the compilation and assembly operations, so is s. A (target-dependent) post-processing operation implemented after the linking procedure is responsible to compute the value s depending on the d_i's and ρ values. Furthermore, the inter-dependencies between checking and checked basic blocks can be resolved at this moment, thanks to some linear algebra over $\mathrm{GF}(2)$.



(a) Before (b) After

Fig. 2: Control flow of `f()` before and after the insertion of a bogus control flow construct

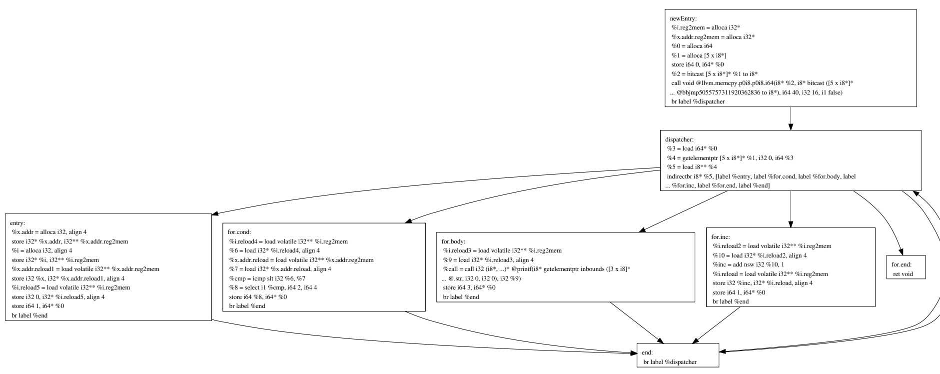


Fig. 3: Call Flow Graph of `f` with Flattening

Finally, the `respond()` routine is performed implicitly. Indeed, an invalid checksum value drives the control flow in an infinite loop in the flattened code, implemented as the `default` case in the `switch` managing the control flow. We note that different responses, such as an early abort procedure, could be called from that default case as well.

G. Procedures Merging

Essentially, the goal of this obfuscation technique consists in merging all functions from a compilation unit (i.e., one `.c` file after pre-processing), into one single unique new function and then replace all calls to the original functions with that new function `merged()`.

More precisely, the mechanism is implemented roughly as follows: the code of each original function is extracted and put in `merged()` as a `case` block of a routing `switch()` structure. Then, each original function is replaced by a simple wrapper function responsible to give the `merged()` function its parameters, in the form of a variable parameters list, as well as a value identifying the original function. Hence, the static information left to the reverse engineer boils down to the signature of the function, as well as to the identifiers linked to it. Since the latter can be computed dynamically, this increases the resistance towards static reverse engineering, as one can figure out only at run-time which part of the code of `merged()` will be executed. The use of wrappers is currently required if several compilation units are involved, in order not to break APIs by modifying the routine names. This problem could theoretically be solved at link time, but this is left for future research.

The resistance of this construction can obviously be improved in a significant way if the `merged()` function is flattened thereafter or modified using other transformations. Another benefit of procedures merging is that it renders more difficult attacks such as code carving, where an adversary extracts some code, without necessity to understand its behaviour, and injects it in another program.

III. Practical Results

Protecting software naturally comes with a price in terms of code size and performances. To illustrate the impact on code size and performances, we have chosen to use the OpenSSL cryptographic library[4], that is implemented in C. Implementations of cryptographic algorithms and protocols

[4]Available through http://www.openssl.org

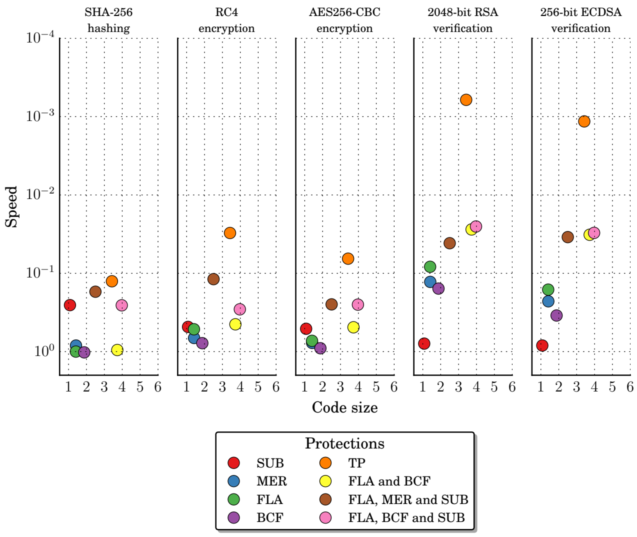


Fig. 4: Summary of the impact on performances and code size for various cryptographic algorithms (SHA256, RC4, AES256-CBC, 2048-bit RSA and 256-bit ECDSA) and combinations of protection techniques.

involve often complicated code that offers great challenges in terms of correctness and performance of code transformations. Fig. 4 illustrates the impact of performance and code size of several obfuscations and combinations of obfuscations on OpenSSL implementations written in C[5] of SHA256, RC4, AES256-CBC, 2048-bit RSA and 256-bit ECDSA. As general remarks, one can note that the code size increases by a factor of 1 to 4. Most simple transformations, such as instruction substitutions, control-flow flattening, or insertion of bogus control flow have a rather reasonable impact in terms of performances, that however becomes more important for algorithms that have a large number of conditional branches, such as RSA or ECDSA. Furthermore, one can notice that the embedding of tamper-proof techniques results in large performance impacts. This is not surprising, as our current implementation requires that each basic block contains at least one `check()` operation, which is terribly expensive in loop-intensive algorithms. Finally, we note that the combination of several techniques results in a performance penalty by a factor of less than 10 in this kind of computationally intensive code.

IV. Future Work

Clearly, the functionalities of `ollvm` can be improved in several ways, which we discuss now. So far, we have concentrated our development efforts on platform-independent code transformations. However, some protections towards reverse-engineering, such as e.g., the insertion of anti-debugging tricks, are dependent of the targeted platform and operating system. It means that one line of further research consists in working at the back-end level, by implementing dedicated generic intrinsic instructions that can be inserted in the middle-end, and that will be translated into machine code able to tightly interact with the underlying hardware and operating system. Another direction that we foresee to work on in a near future is the proper handling of constant values, such as strings, integer constants, or code initializing variables. Such constants most of the time bring a lot of useful information to a reverse-engineer. For instance, various tools exist that are looking for well-known constants used in cryptographic algorithms. Easy transformations can be used to minimize the efficiency of such tools. Another potential future research direction is to transform the inherent code diversification features of `ollvm` into code watermarking capabilities. The number of available transformations for code substitution purposes, and of available opaque predicates can also be significantly increased, to bring a maximal diversity in the produced code. We have implemented so-called code annotation functionalities, that allow a developer to fine-tune the transformations on a per-routine basis. The same has to be implemented for the tamper-proofing mechanism, possibly involving the use of information provided by a profiler. In summary, we believe that the potential research and development opportunities around `ollvm` are countless.

[5] The hashing and encryption operations happen on data of 8192 bytes, while the RSA and ECDSA are public operations, i.e., signature verifications.

V. Conclusion

To the best of our knowledge, Obfuscator-LLVM is the first tool that works at source-code level independently of the programming language and of the target architecture, and having a majority of its source code being open-source and freely available. In its current shape, it should be considered as a preliminary prototype, and we plan to continue its development in a near future. The fact that a large part of its functionalities have been open-sourced allows also to hope for future contributions from third parties. While obfuscation techniques have been widely used by the malware industry since decades, we feel that the availability of such a tool might be beneficial for academic research activities in the domain of software protection and deobfuscation. As a matter of fact, the best industrial-strength obfuscation tools are often difficult or impossible to obtain at a reasonable price for research purposes.

Acknowledgment

The authors would like to thank Stéphane Ongagna for having written a preliminary implementation of procedures merging. Part of this work has been supported by the HES-SO, the Hasler Foundation under contract 12026, as well as the Celtic+ H2B2VS project.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition edition, 2007.

[2] B. Anckaert, M. H. Jakubowski, and R. Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management (DRM'06)*, pages 47–58, 2006.

[3] D. Aucsmith. Tamper resistant software: An implementation. In R. J. Anderson, editor, *First International Workshop on Information Hiding, Cambridge, U.K., May 30 - June 1, 1996*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Aug. 2001.

[5] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In H. Handschuh and A. Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Aug. 2004.

[6] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In Y. Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Mar. 2011.

[7] M. Ceccato, M. D. Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier slicing for remote software trusting. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07), September 30 - October 1, Paris, France*, pages 27–36, 2007.

[8] H. Chang and M. J. Attalah. Protecting software codes by guards. In T. Sander, editor, *Proceedings of the 1st ACM workshop on Digital Rights Management (DRM'01), Philadelphia, PA, USA*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2002.

[9] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Revised Papers from the 5th International Workshop on Information Hiding (IH'02)*, pages 400–414, 2002.

[10] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-box cryptography and an AES implementation. In K. Nyberg and H. M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Aug. 2002.

[11] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[12] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, July 1997.

[13] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages (POPL 1998)*, San Diego, CA, Jan. 1998.

[14] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[15] C. S. Collberg, S. Martin, J. Myers, and B. Zimmerman. The Tigress diversifying C virtualizer. http://tigress.cs.arizona.edu/.

[16] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.

[17] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[18] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[19] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, May 2013.

[20] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, Oct. 2013.

[21] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[22] S. Ghosh, J. D. Hiser, and J. W. Davidson. A secure and robust approach to software tamper resistance. In *Proceedings of the 12th international conference on Information hiding (IH'10)*, pages 33–47, 2010.

[23] B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna. Dymo: Tracking dynamic code identity. In *Proceedings of the Symposium on Recent Advances inIntrusion Detection (RAID), San Francisco, CA, USA*, September 2011.

[24] S. Goldwasser and G. N. Rothblum. On best-possible obfuscation. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 194–213. Springer, Feb. 2007.

[25] L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295. Springer, Aug. 2007.

[26] B. G. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In T. Sander, editor, *Proceedings of the 1st ACM workshop on Digital Rights Management (DRM'01), Philadelphia, PA, USA*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer-Verlag, 2002.

[27] M. Jacob, D. Boneh, and E. W. Felten. Attacking an Obfuscated Cipher by Injecting Faults. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002)*, volume 2696 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2002.

[28] C. Kil, E. C. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Lisbon, Portugal, June 2009*. IEEE, 2009.

[29] P. Larsen, S. Brunthaler, and M. Franz. Security through diversity: are we there yet? *IEEE Security & Privacy*, 12(2):28–35, 2014.

[30] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: automated software diversity. In *Proceedings of the IEEE Symposium on Security and Privacy, May 18-21, 2014, Berkeley, CA, USA*, pages 276–291. IEEE, 2014.

[31] T. László and A. Kiss. Obfuscating C++ programs via control flow flattening. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:3–19, 2009.

[32] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[33] M. Madou, B. Anckaert, B. D. Suter, and K. D. Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital Rights Management (DRM'05)*, pages 75–82, 2005.

[34] F. Monrose, P. Wyckoff, and A. D. Rubin. Distributed execution with remote audit. In *NDSS'99*. The Internet Society, Feb. 1999.

[35] National Institute of Standards and Technology, U. S. Department of Commerce. Announcing the Advanced Encryption Standard (AES). FIPS 197, November 2001.

[36] H. Rajan and M. Hosamani. Tisa: Towards trustworthy services in a service-oriented architecture. *IEEE Transactions on Services Computing*, 1(4):201–213, October-December 2008.

[37] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)*, pages 209–218, 2002.

[38] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[39] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant systems. In *Proceedings of the 8th International Workshop on Information Hiding (IH'06)*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer-Verlag, 2006.

[40] S. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10 pp.–, Nov 2005.

[41] G. W. P. C. van Oorschot and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the IEEE Symposium on Security and Privacy, May 8-11, 2005, Oakland, CA, USA*, pages 127–138. IEEE, 2005.

[42] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, 2005.

[43] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202, July 2001.

[44] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.

[45] B. Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2009.

[46] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277. Springer, Aug. 2007.

[47] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, May 1994.