Securing Embedded Devices through Obfuscation with Predictable Size and Execution Overhead

Leif Brötzmann
Kiel University, Kiel, Germany
lb@bytecode.re

Patrick Rathje
Kiel University, Kiel, Germany
pra@informatik.uni-kiel.de

Olaf Landsiedel
Kiel University, Kiel, Germany
Chalmers University of Technology, Gothenburg, Sweden
ol@informatik.uni-kiel.de

Abstract

Embedded devices compute and store data locally. Thus, contained sensitive data or code requires extra layers of security. In addition to hardware protection, software obfuscation allows for added code and data protection. Available obfuscation techniques, however, are complex, and their resource costs are difficult to predict, rendering them hard to deploy to resource-constrained devices. This holds especially when multiple techniques are combined. This work introduces obfuscation with predictable size and runtime overhead and tailors software obfuscation to the inherent resource limitations of embedded devices. Accurate predictions of size and execution overhead allow dynamic obfuscation utilizing all of the sparse resources. The implemented framework combines several predictable obfuscation techniques with granular control over their parameters, thus allowing precise control over the resulting resource cost. Our evaluations compare our techniques to state-of-the-art approaches, attesting to the precise prediction of our framework. However, the current implementation is best used on small programs or selected parts, showing at least twice the overhead.

1 Introduction

Integrated and deployed on-site, embedded devices typically operate in untrusted environments and inherently expose their hardware and possibly code to attackers. Cryptographic keys, copyrighted material, or proprietary algorithms thus require additional protection. While some platforms provide hardware protection schemes, they are sparsely available and commonly provide specialized protection, like storage for cryptographic keys. However, even on platforms that offer a wide range of hardware mechanisms, additional protection against attacks on the software layer is desirable, as attackers may eventually bypass hardware security. Software Obfuscation aims to harden the software component by disguising the actual instructions that are executed and the data that is operated on. Obfuscation techniques utilize a wide range of approaches, such as scrambling the flow of instructions or hiding calculations using equivalent (but more complex) formulas. In all cases, obfuscation rather hardens than protects the device from attacks, forcing attackers to use more resources and increasing the costs of attacks.

For embedded devices, runtime and size overhead are of great importance. However, existing obfuscation approaches cannot calculate the expected overhead of individual obfuscation runs. Consequently, the obfuscated program can be too large to fit into the device's memory or fail to maintain the timing requirements dictated by the application. This holds in particular for obfuscation methods employing randomness for device-specific obfuscation. Hence, even if one obfuscation meets the requirements, there is no guarantee for its subsequent execution: simple changes to the code might exceed bounds and require reconfiguration. At the same time, free memory could allow for additional obfuscation to strengthen security.

We argue that software obfuscation is a viable method for securing code, especially on embedded devices, yet, the obfuscation needs to be tailored toward the inherent resource constraints. For this reason, we introduce the concept of Predictable Obfuscation, allowing obfuscation techniques to predict bounds accurately. Consequently, our approach enables dynamic combinations and configurations of different obfuscation techniques to maximize security within given constraints. Overall, the contributions of this paper summarize as follows:

1. We establish the concept of Predictable Obfuscation.
2. Our open-source framework implements several techniques targeting ARM-Thumb2 obfuscation.
3. Our evaluation attests to the framework's ability for dynamic obfuscation and compares it to state-of-the-art techniques.

The remainder of this paper continues with the background in Section 2 and related work in Section 3. Section 4 then introduces the concept of Predictable Obfuscation and describes its implementation in Section 5 and Section 6. Section 7 illustrates selected techniques we adjust to the framework and evaluates their resource overheads in Section 8. Section 9 reflects on the results and possible future improvements. Finally, Section 10 concludes this work.

2 Background

Obfuscation describes transforming programs into semantically equivalent programs while trying to hide their inherent secrets. In theory, perfect obfuscation aims for black-box properties [2], i.e., full access to the obfuscated program only provides as much information as access to an oracle. Unfortunately, this property is impossible to achieve in general. In practice, techniques with weaker guarantees are used and studied for their usefulness in hindering attackers. Standard techniques are proposed [1, 6] and result in academic projects like Tigress [4] and Obfuscator-LLVM [11] as well as commercial obfuscation tools.

Unlike traditional compiler passes, obfuscation techniques are not designed to optimize programs for resource cost: For one, they also allow randomization to strengthen the security benefits [17]. Every technique aims to fortify against specific attacks and runs as an obfuscation pass. Hence, combining different techniques, or passes, boosts (in theory) protection against a broader range of attacks. However, optimizing compiler passes is a complex problem, and related research [10] suggests that combining obfuscation passes requires additional considerations: suboptimal pass scheduling may even weaken the resulting security. With the existing tools, predicting the overhead of combined obfuscation techniques is difficult [9].

3 Related Work

While there are plenty of commercial state-of-the-art obfuscators [12, 14, 8] few freely available ones target native code output. Academic-based but closed-source tools like Tigress [4], and open-source Obfuscator-LLVM [11], an obfuscator written for usable and available obfuscation, differ from this work as they do not estimate the resulting overhead. Loki [13], an obfuscator designed to resist state-of-the-art automated attacks, provides experimental evaluations of their overhead compared to other tools but does not introduce any estimated resource boundaries for arbitrary programs. Because of randomization, in practice, overhead varies for different obfuscation runs. So far, research on solving the problem of expensive obfuscation techniques given restricted resources [9] focuses on optimizing the obfuscation process to produce smaller, more efficient, but similarly secure output. However, the converse, i.e., a high level of security given specific resource limitations, remains open.

4 Predictable Obfuscation

The primary idea of this work lies in the prediction of obfuscation passes, allowing precise guarantees about the size and runtime overhead of obfuscation. We introduce this prediction property as *Predictable Obfuscation* and reason about the changes in resource cost caused by the non-deterministic code transformations. Based on our deterministic model using an abstraction of program properties, we derive a methodology for obfuscation, allowing more informed decisions on resource cost and security trade-offs. Finally, we implement the concept of predictable obfuscation with different techniques as a framework for ARM obfuscation.

We aim to reason about the resource cost of applied obfuscation passes concerning a specific metric M such as output size, instructions executed for specific paths, and energy consumption. Therefore, we model the application of several obfuscation passes concerning resource costs. Our abstract model starts with an intermediate representation of a program as input. We denote the set of intermediate representations, i.e., possible programs, with $\mathbb{X}$. We assume that programs halt eventually.

We indicate the set of inputs for programs as $\mathbb{I}$. Given this input, the execution of the program should be deterministic and device-agnostic. Hence, every input $i \in \mathbb{I}$ encodes the actual program input, a randomization seed, and necessary information of the execution environment, e.g., device-specific hardware identifiers.

Based on a metric for resources M, we derive a cost function that measures the inherent resource cost of a program $x \in \mathbb{X}$ under a specific input $I \in \mathbb{I}$ as: $Cost_M \colon \mathbb{X} \times \mathbb{I} \to \mathbb{R}$. Without any prediction, an algorithm can take a program, compile it to native code, and measure the corresponding resource cost under the given input.

However, as we target prediction without the actual compilation or execution, we introduce the concept of properties characterizing a program's execution. We then use those properties, like the number of mathematical operations or basic blocks, to derive its final resource cost. We introduce $\mathbb{P}$ as the set of properties and associate a function $Props_{\mathbb{P}} \colon \mathbb{X} \times \mathbb{I} \to \mathbb{N}^{|\mathbb{P}|}$ that computes property values for a given program and input. For every metric M, we can find a set of properties $\mathbb{P}$ and an associated function $Estimate_{M,\mathbb{P}} \colon \mathbb{N}^{|\mathbb{P}|} \to \mathbb{R}$ such that: $Estimate_{M,\mathbb{P}}(Props_{\mathbb{P}}(x,i)) = Cost_M(x,i)$ holds for all $x \in \mathbb{X}$ and all $i \in \mathbb{I}$, i.e., we find properties and an estimation function such that we can estimate the actual resource cost solely based on the properties. In the most straightforward case, we can directly assign resource costs as property values. Following the motivation, however, the objective is to find properties that encode the resource cost conceptually, capturing the effect of obfuscation passes.

Now within this model, we formalize Predictable Obfuscation: Given a metric M, a set of properties $\mathbb{P}$ and an estimation function $Estimate_{M,\mathbb{P}}$, a tuple $(o \colon \mathbb{X} \to \mathbb{X}, p \colon \mathbb{N}^{|\mathbb{P}|} \to \mathbb{N}^{|\mathbb{P}|})$ is called a *Predictable Obfuscation Pass* if it holds that:

$$Cost_M(o(x), i) = Estimate_{M,\mathbb{P}}(p(Props_{\mathbb{P}}(x,i)))$$

for every $x \in \mathbb{X}$ and every $i \in \mathbb{I}$. Hence, for the obfuscation established by o, the function p encodes the change in its properties' values so that the resource cost can still be estimated. Note that the obfuscation pass does not rely on any particular program input. Because the obfuscated program is again a valid program, we can combine any sequence of Predictable Obfuscation Passes and apply it to our initial program (assuming we find a suitable set of properties and an estimation function).

With our model of predictable obfuscation, we decouple the estimation of resources from the actual obfuscation. A crucial criterion for applicable predictable obfuscation is that the properties, the estimation function, and the obfuscation passes are defined such that the calculation of the costs is less expensive than the actual application of the obfuscation

passes and the execution of the resulting program. For this reason, we introduce algebraic formulas to describe the effect on the program's property values after extracting the initial properties from the input program once. Hence, given the respective predictable obfuscation passes, we efficiently calculate the resource costs of any combination to a respective program.

5 Framework Design

Predictable Obfuscation builds on the idea that we can model the influence of particular obfuscation passes on resource consumption. Hence, the obfuscation can be customized to given constraints based on a mathematical abstraction rather than potentially expensive benchmarking. This section now derives guidelines for realizing our framework based on this theoretic foundation.

Intermediate Representation as Input: All properties of a program that an obfuscation technique may modify are quantified to estimate the overhead of a program. The instruction count and static data for size, the depth of nested loops for runtime, and the type of instructions for energy consumption are exemplary metrics. For this, we generate an intermediate representation of the program that provides all relevant information about the program's properties. The obfuscation passes run on an intermediate representation of the input program. Compared to existing intermediate representation and compilation tooling, our representation is designed with the predictability definitions in mind. The representation thus guarantees the translation process and output of native code follow the abstract properties.

Affine Overhead Formulas: From the fundamental properties of an input program, the framework needs to estimate the change in the properties for each obfuscation pass. Hence, we require exact algorithms that model the influence of modifications on the program's properties. Separating the overhead calculations from the actual application, the framework can efficiently and precisely predict the impact of obfuscation passes. For our framework, we assume an influence that can be modeled by an affine transformation of the property vector, resulting in an efficient estimation process.

Modular Design: Combining several different obfuscation techniques usually hardens the resulting obfuscation. Hence, we design the framework such that the program and overhead calculations are correct regardless of the order and amount of applied obfuscation passes.

Native Format as Output: Resource constraints refer to the final output in native code; our framework must know the final native program to make the overhead calculations meaningful. Thus, the framework further tracks the effect on the final program for each pass based on the properties of the intermediate representation. Finally, the framework outputs native code directly, achieving complete control without any postprocessing that could invalidate overhead calculations.

These requirements have substantial implications for the usability of our approach. As we require input in the form of an intermediate representation, we can only calculate the overhead of compiled programs. Hence, the resulting size depends on the compiler and settings like optimizations or additional compilation parameters. Additionally, the predictions only hold for the compiled output of the framework. The estimations do not cover overheads caused by the file format, i.e., when converting the native code to an executable binary file. Also, when generating higher-level code as output (e.g., LLVM IR), the predictions offer no guarantees for the final native code due to internal optimizations or rewrites inside the respective external toolchains.

In addition, the overhead formulas pose a burdensome requirement: Obfuscation techniques are usually complex by design, and the actual overhead is not ensured to be deterministic. This holds in particular when techniques employ randomness. To achieve deterministic overhead calculations in those cases, the techniques require adjustment such that the changes in the properties of the intermediate representation are always the same.

6 Implementation

We implement our framework in Java and release it as open-source.[1] Implemented using Java, the framework settles with a subset of Java Bytecode from compiled Java programs as input. This bytecode is processed and split into Basic Blocks, which contain intermediate code represented as graphs. The output is native ARMv7 Thumb2 machine code. Wrapper code calling the generated machine code is required to use the output in a program, which we do not consider in the overhead predictions.

7 Predictable Techniques

With this conceptual framework in mind, we adapt well-known techniques to demonstrate how to adjust techniques for the predictable obfuscation framework. Our evaluation then compares those adjustments embedded into our framework to existing obfuscation frameworks. Generally speaking, if the behavior of an obfuscation pass can be expressed correctly using the existing program properties and applied to the intermediate representation, it can be adjusted to our predictable obfuscation framework. However, the predictability becomes less valuable when the deterministic overhead with prediction is significantly larger than the average without any prediction. For one, passes that pad their modifications to match the expected size might introduce a substantial overhead in terms of code size.

In the following, we introduce several techniques that we adapted and implemented in our framework:

Mathematical Operation Encoding: Based on mathematical equivalences from Hacker's Delight [16], mathematical operations are replaced by fixed, more complex expressions. This technique is also known as Encode Arithmetic. Some operations have a list of replacements, of which one is chosen at random each time. While the result of the complex operations is equivalent, they are complicated to simplify for decompiler tooling, rendering the analysis more difficult as the original operations are hidden. Replacing $x+y$ with $x-((\neg y)+1)$ is a simple example. This particular transformation replaces a single mathematical operation with three mathematical operations (addition, subtraction, logical inverse) and one constant (1). We allow prediction for this obfuscation technique by replacing each mathematical oper-

[1] https://bytecode.re/obfuscat

ation with the same number of new operations. As a practical result, we append operations to smaller transformations, matching the transformation with the most operations.

Literal Encoding: Literals encode constant data in a program and often encapsulate critical information. However, literals can be encoded and decoded at runtime. This literal encoding thus makes it harder to identify constants, making static detection of unique values such as magic numbers more challenging. An efficient invertible encoding scheme (e.g., invertible polynomial functions) is chosen to make this technique predictable, which is then used to encode constants at compile time. Then, a copy of the decode code with the same overhead is used for each encoded piece of data. Because every constant is replaced with another constant, the encoded constant, and the decode code, the overhead is always the same for each constant.

Variable Encoding: Like literals, storing variables in an encoded format also hardens the security for static attacks, adding work to analyze their semantics. Encoding variables means that all references to encoded variables require the content of the variables to be decoded before usage. We implement Variable Encoding in the same way as Literal Encoding. Hence, the overhead for each variable store operation is the overhead of the encoding operations, and the overhead for each variable load operation is the overhead of the decoding operations.

Fake Dependencies: Fake dependencies describe the transformation of a term into an equivalent term by introducing additional arbitrary "fake" dependencies. These "fake" dependencies reference non-local values connected to the original term through Opaque predicates [6], which are never true. As a result, independent of the corresponding values' actual content, the original term's semantics are preserved. This technique prevents partial evaluation by decompilation tools, i.e., detecting that a term is entirely local and simplifying it to a constant. This transformation adds "fake" external or global dependencies (global variables, function parameters, environment variables), preventing the original term's simplification from a non-global context. We apply this technique to constant values and hence strengthen other techniques. We enable prediction by using the same transformations on all constants to introduce dependencies to random function parameters or global variables. Through this, the added overhead of the fake dependency is just a load operation per constant and the cost of the chosen predicates.

Bogus Control Flow: The technique of Bogus Control Flow [5, Chapter 4.3.4] introduces new control flows into the program's control flow graph that are never traversed for any input. Hardened variants include adding modified copies of the original control flow to these bogus paths. However, strong opaque predicates are needed to ensure that it is not straightforward to analyze which path is the original. These terms always yield a fixed output independent of their input and are used to decide which path to choose. We apply this technique only to basic blocks with a single successor for simplified overhead calculations. The respective pass turns all the unconditional basic blocks into conditional basic blocks and uses a random predicate from a list of opaque predicates with the same overhead.

Control Flow Flattening: Control Flow Flattening [15] aims to obfuscate the control flow by making all basic blocks predecessor and successor of the same dispatcher basic block. The added dispatcher basic block reenacts the original control flow by acquiring information from each original basic block on which block is supposed to be executed next. Implementing this adds a "next block" variable to the program. At the end of each obfuscated basic block, the block sets the variable to its respective successor block. To prevent using multiple basic blocks to implement conditional "next block" assignments, formulas that achieve the conditional behavior without branching are used instead. Another method [3] changes the "next block" variable relative to the current basic blocks value; this prevents local analyses of which blocks might be the target of a branch. All of these formulas have the same overhead, which makes the overhead of this transformation only dependent on the original program's number of conditional and unconditional basic blocks. To calculate the overhead, we note that the overhead in size and runtime are very different compared to previous cases. For the size of the calculations, the added dispatcher block is only counted once. As for the runtime calculations, the amount of times the dispatcher executions depends on how many basic blocks are executed for the same input in the program without this technique applied.

Virtualization: Virtualization Obfuscation helps to protect code by translating it to a custom architecture instead of its native target. This custom architecture code is then packed together with an interpreter or simulator that decodes and executes it natively. Without adjustments, existing tooling usually fails to analyze the custom architecture code [7]. We consider the compilation process of the custom architecture code and the interpreter for the precise overhead calculations. A straightforward way of implementing this technique is to assemble each intermediate representation node into a single custom instruction with the same length. Then the handlers for each instruction are adjusted for the interpreter to have the same overhead. This way, the size overhead is the original program instruction count times the instruction length added to the size of the interpreter. And the runtime overhead of the original programs overhead times the overhead of the handlers and the dispatcher.

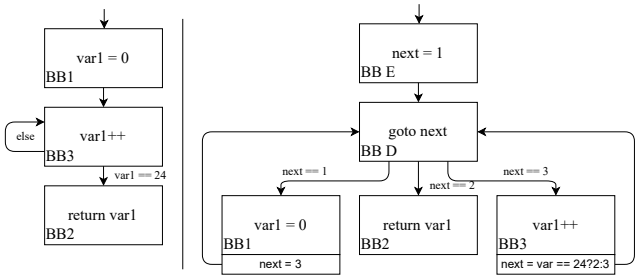

Figure 1. A program that counts up to 24 in its original version (left) and obfuscated version (right) using Control Flow Flattening. Introducing a variable for the next block makes the sequence of blocks harder to analyze.

8 Evaluation

To gauge the proposed framework's overhead, we compare this paper's implemented framework against Tigress and Obfuscator-LLVM. For all our test cases, equivalent programs with similar implementations are used and compiled for the same target. The outputs of our framework are ARM Thumb2 native code blobs linked with a stub compiled with GCC. The Tigress and Obfuscator-LLVM (OLLVM) output both target Thumb2 for ARMv8-A. No optimization flags are provided in either case. The obfuscation passes tested for the comparisons use the same techniques but the details of their implementation differ. The size is calculated directly from the output file size. The instructions executed metrics represent the number of instructions executed for random input strings with a specific length that would traverse the same execution path in the original program. To ensure comparability, we provide relative values. The error bars and values in square brackets denote a measurement's standard deviation.

8.1 Size Overhead

We evaluate the obfuscated program size using two programs: An unoptimized CRC32 algorithm and a SHA1 implementation. Both cases demonstrate how the obfuscators perform on small and big programs. With an unobfuscated, unoptimized GCC compilation baseline, the obfuscators barely cause an overhead for Bogus Control Flow, Control Flow Flattening, and Instruction Substitution on the crc32 program - except for our framework, which causes a 50% size overhead on Instruction Substitution Figure 2.

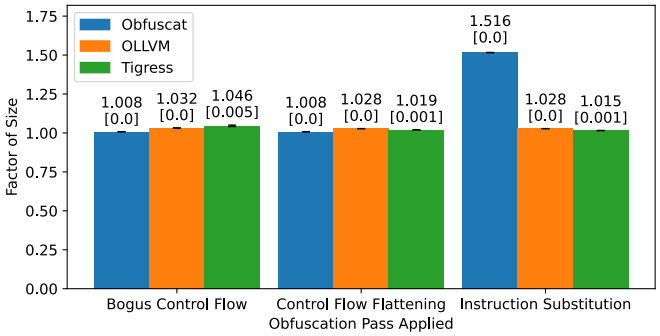

Figure 2. Size Comparison on CRC32: Our framework performs similar to others on small programs

For the more extensive SHA1 program, the overhead of our framework is around three times as much as the comparison obfuscators - again, except for Instruction Substitution, which is 16 times the size of the unobfuscated baseline and significantly worse than the other obfuscators by being five times as much as the OLLVM output and nine times the Tigress output, cf. Figure 3.

Notably, our framework always produces files of the same size. In contrast, the other two obfuscators slightly vary the output size when using different random seeds.

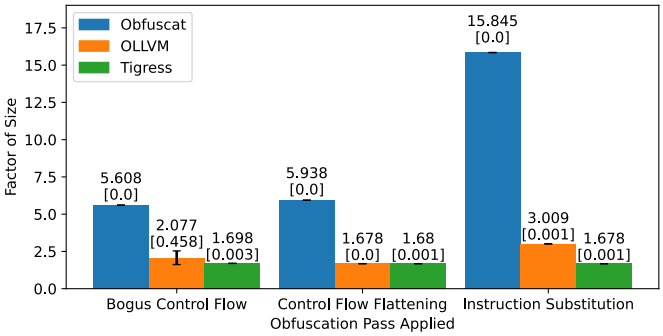

Figure 3. Size Comparison on SHA1: Our framework shows additional overhead on large programs with Instruction Substitution.

8.2 Execution Overhead

For the execution overhead, the obfuscators are again compared on small and large programs. Our framework produces obfuscated code with at least twice to thrice more instructions than the other obfuscators for Bogus Control Flow and Control Flow Flattening. For Instruction Substitution on CRC32, our framework's overhead rises to at least five times the instructions as the other obfuscators, as displayed by Figure 4, and eight times as much on the more extensive SHA1 program (Figure 5).

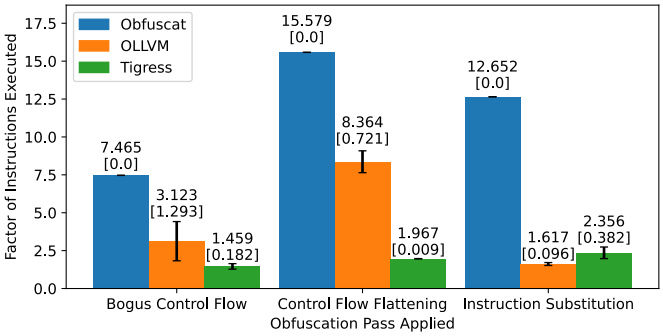

Figure 4. Executed Instructions Comparison for randomized 1024 byte inputs on CRC32: Our framework exhibits additional overhead but remains predictable for each run.

However, our framework consistently produces the same number of instructions, independent of the randomization seed or exact input. This is not the case for the other obfuscators, even though Tigress varies little between the tests.

9 Discussion

The most prominent problem shown by the comparison is the significant overhead. Especially for large programs, our approach causes more considerable overhead than the others. Padding causes most of the unnecessary overhead, which on the native code level, does nothing but fill space. For this work, padding is required to enable accurate predic-

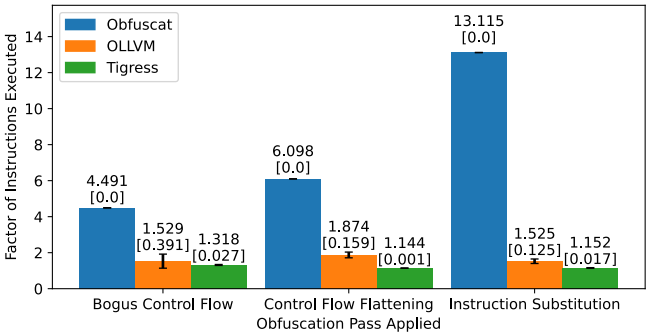

Figure 5. Executed Instructions Comparison for randomized 1024 byte inputs on SHA1: Our framework performs worse but exactly the same each run

tions. However, the efficiency is generally improvable using better compilation models, improving the node-to-assembly translations, or optimizing costly obfuscations. Especially having overhead formulas for compiler optimization passes would be helpful as we then could safely apply optimizations within the predictable obfuscation framework. Alternatively, the prediction could compute only the worst-case overhead, facilitating optimization of the native code while removing the need for padding in the obfuscation techniques. At the same time, the padding could be used for further obfuscation on the assembly level or to replace large paddings with anti-debugging techniques or anti-tampering checks.

Interesting further research is using the predictable obfuscation formulas and resource constraints to solve for possible obfuscation pass combinations using SMT-based approaches or more specialized algorithms. Another open question resides in the calculation of runtime overhead. This work settles on the most predictable unit of executed instructions. However, the concrete number of instructions executed in actual usage scenarios is less critical. Usually, the appropriate characteristic is the overall execution time. While the number of executed instructions and the execution time of a program are related, calculating the exact time from the instruction count on most architectures is not trivial. Precise execution time or energy consumption amplifies the need for more elaborate models. In addition, implementing the Virtual Machine of the Virtualization Obfuscation in a hardware description language and running it on an FPGA is a method that could allow for precise time calculations.

10 Conclusions

In the introduced setting of predictable obfuscation, we derive several techniques with predictable runtime and size overhead. As formulas precisely describe their overhead, optimization algorithms can efficiently derive the best-fitting combination of several passes for given resource constraints without applying resource-intensive obfuscation. Consequently, finding suitable techniques to boost security, including randomization, becomes more available for resource-constrained settings, including embedded devices. We implement predictable obfuscation techniques and bundle them together in our open-source framework. Furthermore, we calculate the resulting binary size and amount of instructions executed for a given input through formulas for each obfuscation technique. Hence, we efficiently calculate the overhead of these techniques, even when combined or randomized. Our evaluation shows the variation of existing tools regarding the actual size and runtime overhead for obfuscations. This variance is significant for some techniques and results in noticeable differences. In comparison, our framework allows precise prediction of the overhead but inherits a significant overhead (often two times as large).

The primary issue in our design and implementation is the additional overhead primarily resulting from the same best and worst-case behavior. Yet, in practice, the worst-case boundaries are the most interesting regarding upper resource limitations. As such, we expect future work to concentrate on the worst-case by removing padding and reducing the worst-case boundary calculations, accounting for more efficient compilation and optimization, and possibly integrating the framework into existing toolchains (e.g., LLVM).

11 References

[1] S. Banescu and A. Pretschner. A tutorial on software obfuscation. *Adv. Comput.*, 108:283–353, 2018.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Electron. Colloquium Comput. Complex.*, TR01, 2001.

[3] J. Cappaert and B. Preneel. A general model for hiding control flow. In *ACM Digital Rights Management Workshop*, 2010.

[4] C. Collberg. The tigress c obfuscator. `https://tigress.wtf`.

[5] C. Collberg and J. Nagra. Surreptitious software - obfuscation, watermarking, and tamperproofing for software protection. In *Addison-Wesley Software Security Series*, 2009.

[6] C. Collberg, C. D. Thomborson, and D. Low. A taxonomy of obfuscating transformations. 1997.

[7] K. Coogan, G. Lu, and S. K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Conference on Computer and Communications Security*, 2011.

[8] Denuvo Software Solutions GmbH. Denuvo Anti-Tamper. `https://www.denuvo.com`.

[9] S. Guelton, A. Guinet, P. Brunet, J. M. M. Caamaño, F. Dagnat, and N. Szlifierski. Combining obfuscation and optimizations in the real world. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 24–33, 2018.

[10] K. Heffner and C. Collberg. The obfuscation executive. volume 3225, pages 428–440, 09 2004.

[11] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[12] Oreans Technologies. Themida – Advanced Windows Software Protection System. `https://www.oreans.com/Themida.php`.

[13] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi. Loki: Hardening code obfuscation against automated attacks. In *USENIX Security Symposium*, 2022.

[14] VMProtect Software. VMProtect Software. `https://vmpsoft.com/`.

[15] C. Wang, J. S. Davidson, J. V. Hill, and J. C. Knight. Protection of software-based survivability mechanisms. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 273–282, 2003.

[16] H. S. Warren. Hacker's delight. 2002.

[17] H. Xu, Y. Zhou, and M. R. Lyu. N-version obfuscation: Impeding software tampering replication with program diversity, 2015.