desync-cc: An Automatic Disassembly-Desynchronization Obfuscator

1st Ulf Kargén
Linköping University
Linköping, Sweden
ulf.kargen@liu.se

2nd Ivar Härnqvist
Linköping University
Linköping, Sweden
ivaha717@student.liu.se

3rd Johannes Wilson
Linköping University
Linköping, Sweden
johwi801@student.liu.se

4th Gustav Eriksson*
Linköping University
Linköping, Sweden
guser908@student.liu.se

5th Evelina Holmgren*
Linköping University
Linköping, Sweden
eveho444@student.liu.se

6th Nahid Shahmehri
Linköping University
Linköping, Sweden
nahid.shahmehri@liu.se

***Abstract*—Code obfuscation is an important topic, both in terms of defense, when trying to prevent intellectual property theft, and from the offensive point of view, when trying to break obfuscation used by malware authors to hide their malicious intents. Consequently, several works in recent years have discussed techniques that aim to prevent or delay reverse-engineering of binaries. While most works focus on methods that obscure the program logic from potential attackers, the complimentary approach of *disassembly desynchronization* has received relatively little attention. This technique puts another hurdle in the way of attackers by targeting the most fundamental step of the reverse-engineering process: recovering assembly code from a program binary. The technique works by tricking a disassembler into decoding the instruction stream at an invalid offset. On CPU architectures with variable-length instructions, this often yields valid albeit meaningless assembly code, while hiding a part of the original code.**

In the interest of furthering research into disassembly desynchronization, both from a defensive and offensive point of view, we have created desync-cc, a tool for automatic application of disassembly-desynchronization obfuscation. The tool is designed as a drop-in replacement for gcc, and works by intercepting and modifying intermediate assembly code during compilation. By applying obfuscation after the code generation phase, our tool allows a much more granular control over where obfuscation is applied, compared to a source-code level obfuscator. In this paper, we describe the design and implementation of desync-cc, and present a preliminary evaluation of its effectiveness and efficiency on a number of real-world Linux programs.

***Index Terms*—Disassembly desynchronization, Code obfuscation, Reverse engineering, x86 architecture**

I. Introduction

Code obfuscation is often used by developers seeking to prevent intellectual property theft, but is also widely used by malware authors as a means to hide their malicious intents, and to delay the development of countermeasures. Studying obfuscation is therefore important both with respect to its legitimate uses for protecting intellectual property, and from the point of view of malware analysts faced with the challenge of reverse-engineering obfuscated malware. Due to the highly practical nature of this field of study, the availability of tools implementing various obfuscation techniques is crucial for furthering code obfuscation research. While there exist a few freely available tools that implement common obfuscation techniques [1]–[3], these, just like most published research on the topic, focus on obfuscation applied at the source-code (or intermediate representation) level, aimed at obscuring program semantics from reverse-engineers. The complimentary technique *disassembly desynchronization* [4], however, has received relatively little attention in the literature, despite being frequently used by malware authors [5]. Disassembly desynchronization puts another hurdle in the way of attackers by targeting the most fundamental step of the reverse-engineering process: recovering assembly code from a program binary. The technique is applicable on processor architectures with variable-length instructions and dense instruction encodings, such as x86, where attempting to decode a machine-code sequence at the wrong offset often yields a valid albeit meaningless assembly listing. By using "fake" branches guarded by opaque predicates (i.e., predicates whose truth value is invariant, but hard to determine statically), it is possible to trick a disassembler into decoding the instruction stream at an invalid offset. This results in hiding a number of original instructions before the disassembly re-synchronizes with the original instruction stream.

To the best of our knowledge, no tools for automatically applying the technique are currently available to the research community. In the interest of furthering research into disassembly desynchronization, both from a defensive and offensive point of view, we have created desync-cc, a tool for automatic application of the obfuscation technique to x86-64 binaries. desync-cc is designed as an easy-to-use drop-in replacement for gcc on Linux that applies obfuscation during compilation. Examples of envisioned use cases for the tool include generation of ground-truth data to aid in the development of new methods for defeating disassembly desynchronization, or as a platform for experimenting with new variations of the obfuscation technique.

*The authors contributed equally to the paper.

In this paper, we discuss important design considerations for a disassembly desynchronization obfuscator, and outline the design and implementation of desync-cc. We also perform a preliminary evaluation on several real-world Linux programs, where we evaluate our tool's efficiency as well as its effectiveness at thwarting code-recovery by a state-of-the-art recursive disassembler.

In the interest of furthering obfuscation research, we make desync-cc available as open source at https://github.com/UlfKargen/desync-cc.

II. Background

Disassembly is the process of translating machine-code instructions into human-readable textual *mnemonics*. While conceptually simple, the problem is in fact undecidable in the general case, due to the possibility of data being interspersed with the machine code. Such data can exist even in non-obfuscated binaries due to, for example, jump tables for switch-case constructs, which are stored inline with the executable code. Simple *linear sweep* disassemblers make a single linear pass over the machine code, and will therefore generally produce incorrect disassembly even for benign binaries with inline data. *Recursive disassemblers* avoid this problem by only performing disassembly along valid (statically deducible) control-flow paths in the binary. Such dissemblers can still, however, be tricked into producing invalid disassembly by means of "fake" branches protected by opaque predicates. This weakness can be exploited to achieve disassembly desynchronization in two ways: the first approach is to insert junk data protected by a branch that will always jump past the data during execution, but whose fall-through edge cannot be statically determined to be unrealizable. An example of this approach is shown in Figure 1. Since the register `eax` is first zeroed using an `xor`-instruction, the branch at offset 2 (pointing to the `call` instruction) will always be taken, so that execution will never reach the junk byte at offset 4. A disassembler that by default processes the fall-through edge of branches first, however, will attempt to disassemble code at offset 4. Since the junk byte (`EB` hexadecimal, highlighted in red) and the first byte of the `call` instruction forms a valid `jmp` instruction (`EBE8` hexadecimal), the disassembly *desynchronizes*, resulting in the rest of the `call` instruction being interpreted as two separate instructions (`je` and `xor`). Since the `xor` instruction ends at the boundary of the original `call` instruction, the disassembly would re-synchronize at that point. This behavior is typical for x86 assembly [4], which means that at most a few original instructions can be hidden by any single desynchronizing opaque predicate.

The second way of achieving disassembly desynchronization is to use a fake branch into the middle of an existing instruction, ensuring that the branch is never taken by means of an opaque predicate. This will cause desynchronization in disassemblers that by default process the taken branch first, given that the instruction and offset is chosen correctly.

Fig. 1. Machine code and corresponding assembly for the cases where the disassembler starts decoding from the taken branch (left) or fall-thorough branch (right), respectively.

```
0.  31C0          xor eax,eax          31C0    xor eax,eax
2.  7402          jz 0x3               7402    jz 0x3
4.  EB            db 0xeb              EBE8    jmp 0xfffffff0
5.  E874563412    call <secret>
6.                                     7456    je 0x60
8.                                     3412    xor al,0x12
```

III. Design and Implementation

Design goals. The overall goal of our work is to automate the process of applying disassembly desynchronization, while giving users granular control of where and how desynchronization points are inserted. In contrast to obfuscation tools that work on the source-code level, we have designed desync-cc to allow code-hiding at the granularity of specific instruction types. This allows, for example, obscuring control-flow instructions to obstruct recursive disassembly.

Stealth, i.e., making it harder for a reverse-engineer to identify which parts of the code that are obfuscated, is not meaningful in the context of disassembly desynchronization. Since overlapping alternative instruction sequences unavoidably constitute a telltale sign of the technique being used, our goal has instead been to make it harder to *automatically* identify which branch edge is the "fake" one, thus avoiding that the obfuscation is removed by means of simple heuristics and scripting.

Design of desync-cc. Necessarily, disassembly desynchronization needs to be applied after the code generation phase. However, applying it directly to binaries would require complex binary-patching, and would not allow accurate liveness analysis of registers (see below). Therefore, the optimal point to apply the obfuscation is at the intermediate assembly-code level, after code generation, but before final assembly into machine code. The selection of appropriate desynchronization-inducing junk byte values, however, must be done on the final binary, since it requires the actual machine code to be known. (For example, the jump offsets of branches is not known until after the final binary has been generated.) Therefore, desync-cc is split up into two *stages*. The first stage, which is applied on the intermediate assembly, inserts opaque predicates and fake branches, and reserves space for junk data by inserting `nop` instructions. The second stage, which is applied to the final binary after linking, selects appropriate values for junk data, and patches those into the binary file.

It is necessary to avoid that side-effects from opaque predicates affect program semantics. In order to avoid unnecessary bloating and runtime overhead from excessive register spilling, we perform liveness analysis to find registers that are safe to use in opaque predicates. If not enough registers are available, the tool can be configured to either use register spilling to free up additional registers (the default), or to simply skip the obfuscation attempt and move on to the following instruction,. Similarly, we also perform liveness analysis on processor flags to avoid unintended side effects. For performance reasons, we

do not perform spilling of flags. This means that desynchronization points cannot be inserted directly before conditional branches or other instructions that read processor flags.

To facilitate liveness analysis, we first construct an interprocedural control flow graph (CFG) from each intermediate assembly file. Since all branch targets are identified with labels in the intermediate assembly, this can be done safely and accurately (in contrast to attempting to do it on the final binary). Interprocedural liveness analysis is then applied within each compilation unit (i.e., each intermediate assembly file). When encountering function calls with statically unknown targets, we take Linux x86-64 calling conventions into account, but otherwise consider all registers as live.

Opaque predicates are supplied as templates with placeholders for registers. Registers of appropriate size are then allocated at each desynchronization point, based on liveness information. The tool ships with a few simple xor-predicates (similar to Figure 1), as well as a set of stronger opaque predicates, which makes use of more complex arithmetic operations. The latter have been borrowed from the LOCO obfuscation tool [1], which is part of the Diablo [6] link-time optimization framework.

When using register spilling, desync-cc is capable of inserting desynchronization points at roughly every second instruction on average. Without spilling, the corresponding figures are around 40% for simple xor-predicates, but only around 5% for the Diablo predicates, due to their more complex nature.

In order to prevent reverse-engineers from trivially identifying which branch edge is the fake one, desync-cc can insert both always-taken and never-taken branches. (For example, if only always-taken branches were used, it would be trivial to deobfuscate a binary by replacing every desynchronization-inducing branch with an unconditional jump.) Symbols storing the offset of each predicate are inserted into the binary, to allow the second stage to locate patch-points. In addition to a unique identifier, the number of inserted junk bytes are encoded as part of the symbol name. (The number of junk bytes is a configurable parameter.) The second stage then processes each symbol. Patch-points are processed in reverse order based on their offset in the binary. This is necessary since forward traversal might change the desynchronization behavior of previous desynchronization points when applying patches (if, for example, the desynchronized stream overlaps with the junk bytes of the next patch-point).

In order to find appropriate junk bytes for the always-taken branches, we read, for each symbol, a chunk of code (50 bytes by default) starting from the symbol's offset (i.e., the beginning of the junk data), and disassemble it. We then replace the initial junk-byte portion of the code chunk (which is initially filled with `nop`s) with random data and disassemble the result. If no instruction starts at the offset where the junk-bytes end, we have successfully achieved desynchronization. Otherwise, we repeat the above until successful desynchronization or an iteration limit is reached. To prevent trivial identification of fake branch edges, we also consider the desynchronization a failure if it results in invalid instructions.

For never-taken branches, we select a jump-offset in the same range as the junk-data size (to make never-taken branches harder to distinguish from always-taken ones). For each offset in the range that are not on an original instruction boundary, we attempt disassembly over a fixed window (50 bytes). Similarly to the above, desynchronization is considered successful if the disassembly does not result in invalid instructions. Since it is not always possible to find such offsets, we select a valid instruction boundary as a fallback (or a random offset as last resort). Finally, the never-taken branch instruction's target is patched to the chosen offset.

Implementation. We implement desync-cc using gcc's `-wrapper` command-line option, which invokes each of the gcc subcommands (for compilation, assembly, linking, etc.) under a wrapper program. Our wrapper consists of a shell script that parses the subcommand command-line and invokes the appropriate desync-cc stage. Stage 1 is implemented in about 3,000 lines of C++ and stage 2 in about 300 lines of Python. Stage 1 makes use of the Keystone and Capstone libraries[1] for assembly and disassembly, respectively. During CFG construction, instructions are first assembled using Keystone, and Capstone is then used to extract details on which registers and flags that are read or written. This information is then used for liveness analysis. Stage 2 makes use of the Python bindings for Capstone for disassembly. Since the vast majority of processing time is spent on repeated disassembly, the use of Python does not have a big impact on performance, as the disassembler is implemented in native code.

Usage. To apply obfuscation using desync-cc, the user simply sets the `CC` variable to the desync-cc executable when running `configure` for an Automake project, or `make` for a simple makefile project. desync-cc uses a configuration directory containing a configuration file and a subdirectory with one or more predicate template files. It is possible to override the default configuration by setting the environment variable `DESYNC_CONFIG_BASE_DIR` to the path of a custom configuration directory. The configuration file allows granular control over the obfuscation process. For example, it is possible to use a regular expression to control which instruction types the obfuscator attempts to hide, the frequency at which predicate-insertion is attempted in the code, the junk byte length (fixed, range, or Gaussian distribution), the fraction of branches that should be of the never-taken type, etc.

desync-cc leaves the desynchronization-point symbols in the executable to serve as ground truth for experimentation, but these can easily be removed using the `strip` command from the GNU development tools.

IV. Evaluation

In this section we present a preliminary evaluation of desync-cc. An Intel Xeon E3-1245 (v1) machine with 16GB RAM was used for the experiments.

In our first experiment, we measured the time overhead that obfuscation added to the build process when building the

[1] https://www.keystone-engine.org/, https://www.capstone-engine.org/

109 programs in the GNU Coreutils suite[2]. We configured desync-cc to attempt predicate-insertion every 5 instructions, using simple xor-predicates, with the default configuration of an equal split of always-taken and never-taken predicates. We performed ten runs for each junk-byte size in the range 1–7 bytes. The same random seed was used for all runs, ensuring that predicates were inserted at the same code locations in each of the runs.

The mean results for the ten repeated runs are presented in Table I. The time overhead is shown relative to the non-obfuscated build time, which was 60.13 seconds (σ = 2.07s). We see that the slowdown is around 4x for most cases. It is possible to find desynchronization-inducing junk bytes in almost all cases when using always-taken branches. The low average number of trials needed also indicate that such junk bytes are almost always found quickly, even with our simple brute-force approach. Using more than one junk byte appears beneficial, as is evident from the lower average number of trials, and that the 1-byte runs were the only ones where some desynchronization attempts failed. With an increasing number of junk bytes, the mean number of trials increase slightly. This is likely because it increases the likelihood of introducing invalid instructions within the junk bytes themselves. For the never-taken branches, there is less opportunity to achieve desynchronization, since we are limited to a jump a few bytes ahead into adjacent instructions. Nevertheless, desynchronization is successful in most cases. Since we use the junk-byte length as the upper limit for the jump offset of never-taken branches, using more junk bytes increases the likelihood of success.

We also wanted to evaluate the effectiveness of the obfuscation, as well as the overhead when running obfuscated programs. Since the programs in GNU Coreutils are quite simple and generally execute very quickly, they are less well-suited for evaluating runtime overhead. Instead, we used the XML parser `xmllint` from Libxml2[3], and the compression tool `xz`[4] for this experiment. For `xmllint`, we measured the time required for running its test suite, whereas for `xz` we created a simple benchmark consisting of compressing 30MB or random data.

Since recursive disassemblers discover code in the executable by traversing the control-flow and call graphs, an effective way to hide code from such disassemblers is to obfuscate branching instructions. Therefore, we configured desync-cc to attempt desynchronization close to different kinds of control-transfer instructions. Here, we used the stronger (and more computationally expensive) predicates from the LOCO/Diablo obfuscator. Additionally, we configured desync-cc to only use always-taken branches, since recursive disassemblers typically follow the fall-through edge first.

We used the freeware version of the state-of-the-art recursive disassembler IDA[5], and compared its ability to identify functions in the binaries, with and without obfuscation. Table II summarizes the results. Evidently, it is sufficient to just obfuscate `call` instructions (the second row in the table) in order to prevent IDA from discovering the vast majority of functions. In fact, almost the entire code section of the binary is detected as "data" by IDA when inserting desynchronization points before every `call` instruction. The runtime overhead (calculated as the mean of 10 runs) is quite modest (10–15%) for `xmllint`, whereas the difference for `xz` is so small that it falls within the margin of error. Obfuscating both `call` and `ret` instructions does not result in any additional reduction in function recovery, but slightly increases the overhead.

We also measured the result of obfuscating other branch instructions, namely unconditional jumps (`jmp`), in combination with the comparison instructions typically executed just prior to a conditional branch (`cmp` and `test`). (Note that we cannot place a desynchronization point directly before a conditional branch, since the non-liveness constraint on processor flags would never be satisfied.) This also results in a marked reduction of identified functions, although less so than `call` obfuscation. In this case, IDA was able to correctly identify a larger portion of the code as instructions, but still failed to determine function boundaries in most cases. The overhead in this case was larger (around 100% for `xmllint` and 30% for `xz`.) The last row shows the case where all the five instruction types are subject to obfuscation. This does not result in an improvement over the `call` case in terms of code hiding, but further increases the overhead a bit.

V. Related Work

Linn and Debray [4] were the first to introduce the concept of disassembly desynchronization. They proposed a technique for automatic application of the obfuscation to binaries, but their approach has several limitations compared to ours. Most notably, junk bytes can only be inserted at the beginning of basic blocks preceded by an unconditional branch. They also select junk bytes as subsequences of a fixed short byte-sequence, which would make detection of fake branch edges relatively easy.

LOCO [1] is an open-source obfuscation tool for aiding in manual obfuscation of binaries. The tool supports a few common obfuscation techniques like control-flow flattening and insertion of opaque predicates for obscuring program logic (but not disassembly desynchronization). While the tool allows fine-grained control of where obfuscation is applied, it does not support automatic obfuscation. Instead, it provides a GUI for aiding in manually inserting, e.g., opaque predicates.

Obfuscator-LLVM [2] is a more recent obfuscator that can automatically apply transformations like opaque predicates, bogus control flow, and control-flow flattening during compilation. Since it works on the LLVM intermediate-representation level, it cannot directly be used to apply disassembly desynchronization. Obfuscator-LLVM is available as open source.

The Tigress obfuscator [3] applies transformations at the source-code level to C programs, and supports several advanced obfuscation techniques, such as virtualization-

[2] https://www.gnu.org/software/coreutils/
[3] http://xmlsoft.org/
[4] https://tukaani.org/xz/
[5] https://hex-rays.com/ida-free/

TABLE I
MEAN OBFUSCATION OVERHEAD AND DESYNCHRONIZATION SUCCESS-RATE WHEN BUILDING GNU COREUTILS

Junk bytes	Slowdown	Mean junk byte trials	Successful always-taken branches (out of 59,150)	Successful never-taken branches (out of 58,044)
1	4.70 (σ = 0.83)	4.01 (σ = 0.00)	59,113.0 (σ = 0.0)	-
2	4.18 (σ = 0.13)	2.04 (σ = 0.01)	59,150.0 (σ = 0.0)	44,921.1 (σ = 1.4)
3	4.12 (σ = 0.12)	1.76 (σ = 0.01)	59,150.0 (σ = 0.0)	48,677.7 (σ = 0.9)
4	4.14 (σ = 0.21)	1.73 (σ = 0.01)	59,150.0 (σ = 0.0)	50,585.1 (σ = 0.9)
5	4.24 (σ = 0.31)	2.14 (σ = 0.01)	59,150.0 (σ = 0.0)	53,751.6 (σ = 1.4)
6	4.25 (σ = 0.31)	2.15 (σ = 0.01)	59,150.0 (σ = 0.0)	54,396.0 (σ = 1.0)
7	4.27 (σ = 0.31)	2.20 (σ = 0.01)	59,150.0 (σ = 0.0)	56,160.9 (σ = 1.1)

TABLE II
EXECUTION TIME AND IDA FUNCTION RECOVERY FOR `XMLLINT` AND `XZ`

Configuration	`xmllint` Time (s)	`xmllint` Func.	`xz` Time (s)	`xz` Func.
No obfuscation	7.89 (σ = 0.07)	1762	10.92 (σ = 0.11)	484
`call`	8.89 (σ = 0.11)	9	10.80 (σ = 0.34)	9
`call,ret`	9.98 (σ = 0.08)	9	11.58 (σ = 0.25)	9
`cmp,test,jmp`	15.84 (σ = 0.22)	287	14.27 (σ = 0.38)	27
All branch	17.60 (σ = 0.19)	9	14.41 (σ = 0.44)	9

obfuscation and self-modifying code. Tigress supports insertion of opaque predicates in combination with junk-byte blocks. This can be used to achieve disassembly desynchronization [7], but since transformations are applied on the source-code level, it is not possible to have granular control over where desynchronizations happen. Tigress is publicly available, but source code is available only on demand.

Since desync-cc works on the assembly level, its capabilities are orthogonal to those of tools like Obfuscator-LLVM or Tigress. Our tool can therefore be used to add another layer of protection by applying disassembly desynchronization on top of the protections added by a source-code level obfuscator.

VI. FUTURE WORK

There exist many possibilities for future additions to desync-cc. For example, the currently used opaque predicates are not very resilient against a skilled human analyst, or advanced automated analysis methods, such as symbolic execution [8]. Stronger opaque predicates, which make use of complex data structures or are constructed across multiple functions [9], must be applied during compilation, and are therefore not directly applicable in desync-cc. One possible solution to this problem could be to store the truth-value of strong source-code level predicates (such as the ones provided by Tigress) in global variables, which could then be accessed by desync-cc when applying obfuscation at the assembly-level. This approach could also reduce the runtime overhead of desync-cc, by enabling compile-time optimization of predicates, in addition to the possibility of "reusing" a predicate evaluation at multiple desynchronization points.

In our current implementation, we select junk bytes randomly. An interesting topic of future work would be to explore ways of optimizing the selection of junk bytes. One option would be to optimize with regards to the *quantity* of desynchronization, i.e., to maximize the distance until resynchronization occurs. Another interesting avenue of future work would be to optimize with regards to the *quality* of desynchronized code, i.e., to make it less easily distinguishable from "real" code by a human analyst.

Other potential future improvements could include even more granular control of obfuscation, such as the possibility to chose different predicates for different instruction types. This would, for example, allow inserting always-taken branches before each `call` instruction to confuse recursive disassemblers, while also inserting never-taken branches at random places to make automatic deobfuscation harder.

VII. CONCLUSION

In this paper we have described the design and implementation of desync-cc, a tool for automatic application of disassembly desynchronization. Our preliminary evaluation showed that it incurs a slowdown of around 4x during compilation, and demonstrated its effectiveness at thwarting function recovery in a state-of-the-art disassembler while introducing a modest runtime overhead. We make the tool available as open source in the interest of furthering research into disassembly desynchronization.

REFERENCES

[1] M. Madou, L. Van Put, and K. De Bosschere, "LOCO: An interactive code (de)obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2006, p. 140–144.

[2] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 3–9.

[3] (2021) The Tigress C obfuscator. [Online]. Available: https://tigress.wtf/

[4] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003, p. 290–299.

[5] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. No starch press, 2012.

[6] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, 2005, pp. 7–12.

[7] Y.-J. Tung and I. G. Harris, "A heuristic approach to detect opaque predicates that disrupt static disassembly," in *Binary Analysis Research Workshop (BAR)*, 2020.

[8] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning," in *26th USENIX Security Symposium*, 2017, pp. 661–678.

[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.