Obfuscation of function block diagrams

Antti Pakonen

Citation:

A. Pakonen. Obfuscation of function block diagrams. 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA), Sinaia, Romania, September 12-15, 2023. IEEE, 2023.

DOI: 10.1109/ETFA54631.2023.10275363

Obfuscation of function block diagrams

Antti Pakonen
VTT Technical Research Centre of Finland Ltd., Espoo, Finland
Email: antti.pakonen@vtt.fi

***Abstract*—Obfuscation is a process of transforming a program into an equivalent version which is harder to understand and reverse-engineer. Little attention has been paid to obfuscation techniques for programs written for programmable logic controllers (PLC). However, there is no reason to assume that an attacker would not be interested in hiding malicious payload into a PLC program before it is compiled to machine code.**

In this paper, I present five techniques for obfuscating IEC 61131-3 Function Block Diagram (FBD) programs. Four of the techniques are specific to the graphical representation of FBD. I then evaluate the applicability of each technique by experimenting with different PLC programming tools. I prove that at least four of the techniques are practically applicable, and demonstrate features that some tools successfully use to prevent abuse. Stricter rules, if implemented in IEC 61131-3, would prevent some of the techniques listed.

***Index Terms*—Programmable logic devices, Embedded software, Cyberattack, Intellectual property**

I. Introduction

Software obfuscation can be used for good or evil.

Software developers can protect their intellectual property by making their code harder for a pirate to reverse-engineer [1]. In a cybersecurity sense, obfuscation makes it more costly for an outside attacker to understand and tamper with the program they have gained access to [1], [2].

On the other hand, a skillful attacker can employ obfuscation to hide their tracks and thwart discovery after injecting malicious code [3].

Attackers have been able to inject malware into industrial programmable logic controllers (PLC) in critical applications (see, e.g., Stuxnet [4] or TRITON [5]). While the target is typically the code in the PLC memory, we should not assume that a bad actor would not be interested in hiding malicious code in a PLC program before it is compiled, and the attack can also come from inside the developer organisation.

Accordingly, to cover all aspects of cybersecurity and intellectual property protection, the industry should be aware of the possible ways to obfuscate PLC programs written in the most common languages—the programming languages specified in the IEC 61131-3 standard [6].

However, the literature on obfuscation techniques has to great extent focused on either compiled machine code, or general-purpose languages like C/C++ and Java/Javascript [3].

In this paper, I present five techniques for obfuscating IEC 61131-3 function block diagram (FBD) programs. Four of the techniques are novel, and specific to FBD programs. For the well-known [3], [7] technique of "Garbage insertion", I specify FBD specific patterns for adding superfluous diagram parts. I then evaluate the applicability of each technique by experimenting with five different PCL programming tools.

Knowledge of the techniques may be of use for:

- I&C engineers in protecting their intellectual property, or making their programs more difficult to tamper with,
- FBD programming tool vendors in developing features in their tools to detect inadvisable programming practices,
- cybersecurity experts in bracing against attack types, and
- IEC Technical Committee (TC) 65 in further clarification of requirements in the next edition of IEC 61131-3.

In the sections below, "the standard" refers to IEC 61131-3:2013 (Edition 3.0) [6], and "annex" refers to the supplementary data online[1].

II. Preliminaries

A. Software obfuscation

Obfuscation is the process of transforming a program into a semantically equivalent version which is harder to reverse-engineer [1]. It adds a layer of defence by increasing the effort and the cost for an attacker to gain an understanding of the program functionality [3].

Different techniques are listed and classified in, e.g., [1], [3], [7]. Following [1] and [3], we can classify the transformations into three categories:

1) *Control flow transformations* aim at altering the flow of the program. Two commonly [3] used techniques are garbage (or "bogus") insertion and the use of opaque predicates (both of which are employed in Section IV-C).
2) *Data transformations* include variable splitting and merging, array restructuring, class transformation, class hierarchy flattening, etc.
3) *Layout / lexical transformations* consist of scrambling identifiers, removing comments, reformatting the source code, etc.

In their literature review [3], Hosseinzadeh et al. found that most studies on software obfuscation focused on one or two specific languages, with C/C++ and Java/JavaScript representing the vast majority. Still, if compiled binary code were considered one "language", it would have been the topic of most studies [3].

The popularity of the C language as the target of obfuscation is also exemplified by the International Obfuscated C Code Contest[2], held semi-regularly since 1984.

[1] https://doi.org/10.5281/zenodo.8023237
[2] https://www.ioccc.org/

B. IEC 61131-3 FBD

The IEC 61131-3:2013 is a globally adopted standard for PLC programming languages [6]. Non-standard, vendor-specific languages are still used in specific contexts (e.g., nuclear power plant safety I&C [8]), but overall, popularity of 61131-3 is evident. It is the "leading paradigm" [9] in industrial control.

The standard specifies five standard programming languages:

1) Function Block Diagram (FBD)
2) Instruction List (IL)
3) Ladder Diagram (LD)
4) Sequential Function Chart (SFC)
5) Structured Text (ST)

In FBD, the logic is programmed graphically, by connecting signal flow lines between rectangular blocks to form a network. Each block represents a certain function, having its input parameters on the left, and output parameters on the right side. The line connections describe how data flows from block outputs (or input variables) to block inputs (or output variables). Additional EN inputs and ENO outputs can be used for controlling the execution of the blocks.

Section 6.6.2 of the standard specifies standard functions common to all the languages (e.g., AND, OR, XOR, SIN, COS, TAN, MAX, MIN, ...), but programmers are free to specify their own functions, and the internal logic of the function can be expressed in any of the five languages.

Section 8 specifies rules for the use of graphical elements in FBD and LD, including the representation of lines and blocks, direction of flow, and evaluation of networks.

Many details about logic processing, error handling and graphical representation are left as "implementer specific" decisions, which will prove relevant for some of the techniques proposed in Section IV.

III. Related research

Based on a literature review, obfuscation of 61131-3 FBD programs is a previously unexplored topic.

In [10], the obfuscation is targeted at the compiled code running on the PLC, in order to prevent decompilation of the code back to the FBD program, therefore delaying the investigation of the attack. In this paper, the focus is on the obfuscation of the FBD program before it is compiled.

In, e.g., [11] and [12], the obfuscation is also applied to the compiled code running on the PLC.

In [2], Yang et al. propose three obfuscation techniques for Statement List (STL), a programming language for Siemens' SIMATIC system, but claim that the techniques can be extended to other IEC 61131-3 based languages. First, the "Textual substitution" technique of identifier scrambling makes a program harder to read, but also makes the obfuscation immediately obvious to the reader. Second, the "Control flow scrambling" technique relies on jump instructions. The standard does specify a graphical elements for an unconditional or conditional jump in FBD, and while such jumps are more common in IL, ST or LD, the abuse of jumps in FBD could be a topic for further consideration. Third, the "Redundancy enhanching" technique contains well-known ideas also used in Section IV-C.

In [13], Lopes presents a tool for extracting sequential logic from LD programs to translate the program to SFC. Lopes notes that the translation "somewhat" obfuscates the sequential behaviour.

Some techniques developed for logic circuit obfuscation (e.g., [14], [15]) could be applicable to LD programs, in particular.

IV. Obfuscation techniques for FBD

In this section, I present five obfuscation techniques for FBD programs. The first is a type of lexical transformation, while the other four are control flow transformations.

A. Non-Latin characters in identifiers

According to section 6 of the standard, the characters used in, e.g., function identifiers, are "represented in terms of the ISO/IEC 10646". This provides us with interesting opportunities, since, for example, the Cyrillic capital letter Р (Er) is pretty similar to the Latin P, as is the Greek capital letter Ρ (Rho). We could therefore take a function like EXPT, and then specify an altogether different function like EXPT[3] (spelled with an Er). Just by looking at the diagram, the reader could then be fooled into thinking that an element that looks like the exponentiation function is just that.

Consulting the ISO/IEC 10646 [16], we can find simulacra for almost the all Latin characters (see Table I for select examples, the full list is in the annex), only lacking suitable replacements for F, G, Q, R, U, and W. Since no function defined in IEC 61131-3 is spelled using only those characters, we can create a simulacrum block for every default function there is.

Table I
Potential simulacra for Latin letters K, L and M in ISO/IEC 10646

Letter	Potential simulacrum	Code point
...	...	...
K	GREEK CAPITAL LETTER KAPPA	039A
К	CYRILLIC CAPITAL LETTER KA	041A
K	KELVIN SIGN	212A
L	ROMAN NUMERAL FIFTY	216C
M	GREEK CAPITAL LETTER MU	039C
М	CYRILLIC CAPITAL LETTER EM	041C
M	ROMAN NUMERAL ONE THOUSAND	216C
...	...	...

The blocks with the fewest potential simulacra are SR and RS, since there are no simulacra for the letter R, and only Ѕ (CYRILLIC CAPITAL LETTER DZE, code point 0405) to replace S.

The function with most potential simulacra (1457) is REPLACE, so fittingly, we could use simulacra of REPLACE to replace quite a large library of function blocks (see Fig. 1).

[3]Er sticks out here, due to LaTeX rendering the letter using a different font.

Other functions with over a hundred potential simulacra are CONCAT (485), DELETE (323), LIMIT (215), EXPT (107), INSERT (107) and SPLIT (107).

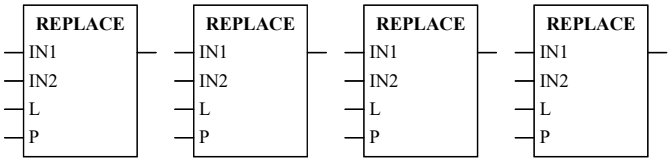

Figure 1. Four different blocks, none of them being the IEC 61131-3 REPLACE. (The figure is a PDF file, so the reader can verify this by, e.g., copying the identifiers to a text editor.)

Whether this works in practice, of course, depends on the font, and the programming tool accepting such characters in identifiers.

In Times New Roman, for example, its fairly clear that the CYRILLIC CAPITAL LETTER KA is different from the Latin K. (But since K is not used in the name of any function defined in IEC61131-3, this is irrelevant.) Otherwise, the character symbols are effectively the same. With other fonts, this might not be the case.

B. Logic hidden in execution control parameters

In FBD, is possible to use additional Boolean EN input and/or ENO output for controlling the execution of each block. Would it be possible to hide the actual logic we want to specify in the processing of these execution control variables?

If the EN input is FALSE, the ENO output will always set to FALSE. This means that we cannot construct an OR gate, but we can still achieve functional completeness (and build any logic function) if we can construct both an AND and a NOT.

To build the AND, we have to use block inputs other than EN to set the block to error state at our will, which will cause ENO to go FALSE (even if EN is TRUE).

Here, we try building our basic logic components using only standard functions defined in IEC 61131-3. The standard specifies that a division by zero is an error, so we can create a logical AND (in the ENO) by converting the signal to INT type and then using the DIV block (see Fig. 2). Depending on the implementation, MOD might work too.

The standard also specifies that it is an error, if the evaluation of a character string function results in an attempt to "access a non-existent character position in a string". The character positions "shall be considered to be numbered 1, 2, ..., L", so "0" would cause an error. We can therefore construct an AND similar to the DIV variant by using the P input of block MID, INSERT, DELETE or REPLACE (see Fig. 2).

We then need a NOT, but let us not go with the most obvious choice of "O" negation in either the EN input or the ENO output. We can use the same blocks we used for the AND, but we do have to negate the input (see Fig. 3).

It might also be possible to achieve all of this with just one standard block. It seems that the standard would allow us to use a multiplexer (MUX) block with just one input (IN0)

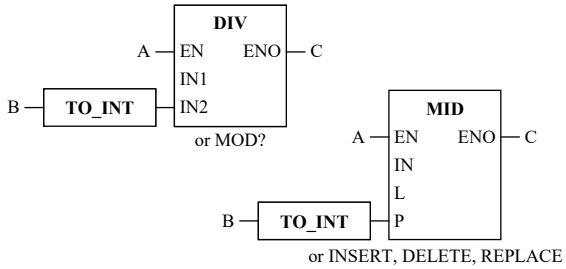

Figure 2. AND function implemented in the EN/ENO logic

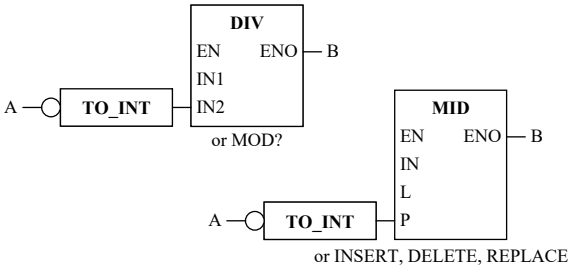

Figure 3. NOT function implemented in the EN/ENO logic

to select from. If the K input is not within the range—the range now being $\{0\}$—we cause an error. Since K and IN0 are of type ANY_ELEMENTARY, we can just connect a BOOL signal to each input. This allows us to construct both a NOT and then an AND (using the NOT logic), giving us NAND (see Fig. 4).

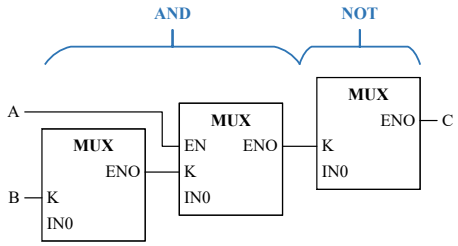

Figure 4. NAND function implemented in the EN/ENO logic of MUX

NAND is functionally complete, so we can now construct any logic function we want using just our NAND, so that we only ever use the (one-input) MUX, and only ever read the ENO output. To illustrate the idea, Fig. 5 shows a (two-input) multiplexer using the construction $[in0\,\mathbf{NAND}(k\,\mathbf{NAND}\,k)]\,\mathbf{NAND}(in1\,\mathbf{NAND}\,k)$ [17].

In nuclear power plant I&C systems, the safety I&C platforms Teleperm XS (by Framatome) and Spinline (by Rolls-Royce) both assign a "fault" status to each *signal* (rather than block) on the diagram [8]. Upon detected failures of measuring equipment, the associated data can then be marked invalid, and excluded from voting logics. Could we hide the actual logic in the processing of the "fault" statuses?

Unfortunately, there is very little information to share. It is public that in Teleperm XS, "AND logic and OR logic use

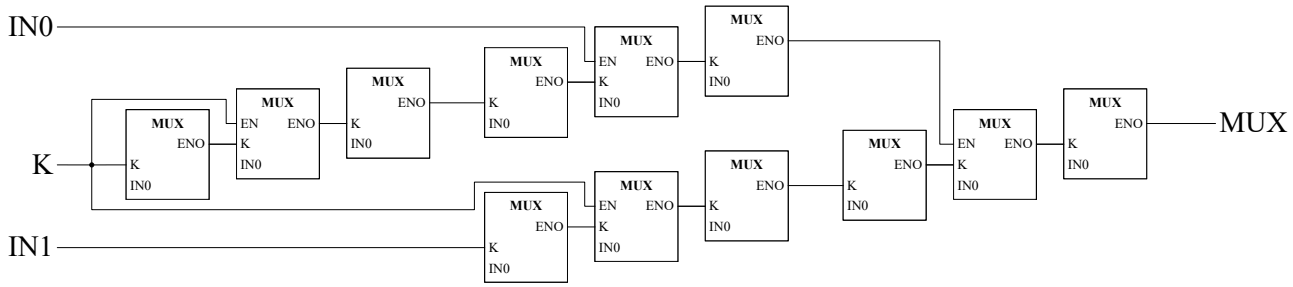

Figure 5. A two-input multiplexer implemented in the EN/ENO logic using four NANDs based on MUX (with one input)

passive status processing. That is, if one input is invalid, the output is invalid regardless of the status of the remaining inputs" [18]. So, an OR operation is inherent in "passive" blocks. Whether a NOT can be constructed using blocks with "active" status processing—to achieve functional completeness—is left as an exercise to a reader with access to the right documents.

C. Garbage insertion

If superfluous logic does not come with a computational cost significant for the application, one way to obfuscate a block diagram is to insert unnecessary garbage [3], [7]. When adding such clutter, one has to make sure that the extra "fake" logic we add has no effect on the diagram outputs we care about.

The figures below introduce exemplar patterns for excluding superfluous signals from the actually desired processing logic. Two of them result in a Boolean expression that always evaluates to FALSE, which is an example of an *opaque predicate* [1], a widely [3] used control flow obfuscation technique.

Such patterns should not be obvious to notice. The patterns below are not my inventions, but taken from real, practical industry projects [8], [19], where I have found the issues using *model checking* [20]. Here, the logics are simplified and their origin is masked.

First, in Fig. 6, the bistable SR is always set when the signal connected to its S1 input is TRUE. If the same signal is FALSE, the negation causes the SR to reset, regardless of the lower OR input. This allows us to connect whatever confusing logic to the OR block, while letting the actual signal pass through unchanged.

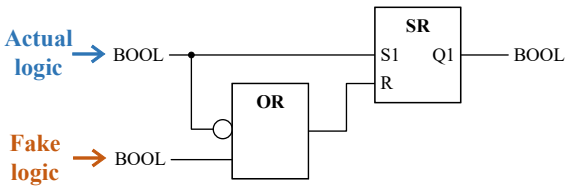

Figure 6. Any signal or logic connected to the lower OR input has no effect on the bistable SR output.

Second, in Fig. 7, the TON (on-delay) would need to receive a input signal lasting more than five seconds to active, but the signal pulse from the TP always lasts exactly five seconds. Therefore, the output of the TON is always FALSE. We can connect whatever logic to the TP, and the actual signal will always pass through unchanged. (We also used this logic in [21] to prove certain points.)

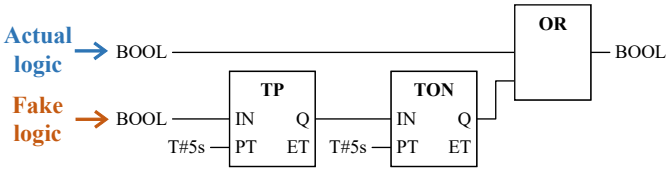

Figure 7. The output of the TON (on-delay) is always FALSE, so any signal or logic connected to the TP (pulse) has no effect on the OR output.

Finally, in Fig. 8, the logic is seemingly comparing three real variables, to check if the values differ from one another. In a kind of majority vote, the low limit of 4.0 is compared against the second-smallest (rather than smallest) value, and the high limit of 5.0 to the second-largest value[4]. However, when selected out of three values, the second-smallest and second-largest are always the same, which cannot simultaneously violate both limits. As the AND is therefore always FALSE, we can connect any kind of confusing logic to the real number inputs, and the actual signal will always pass through unchanged.

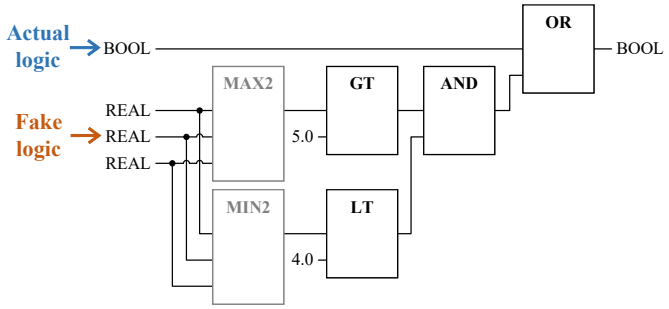

Figure 8. The output of the AND is always FALSE, so any signal or logic connected to the MAX2 and MIN2 blocks has no effect on the OR output.

[4]The second minimum (MIN2) and second maximum (MAX2) are not standard functions defined in IEC 61131-3.

D. Masking in the diagram

Masking is a technique where a function block is hidden from view by placing it behind another function block. By suitable placement, it could then be possible to connect the hidden block so that the connections would seem to be wired to the block on top.

The standard does not seem to contain any rule or requirement against stacking the blocks on top of one another. According to section 8.1.4 on representation of lines and blocks, "[any] restrictions on network topology in a particular implementation shall be expressed as Implementer specific."

E. Crossing signal lines

According to section 8 of the standard, the use of letters or graphic to represent lines connecting the blocks is "implementer specific and not a normative requirement". Lines "can be" extended by using a connector. And, again, any restrictions on network topology "shall be expressed as Implementer specific."

In other words, the standard allows the connection lines to overlap or cross in potentially ambiguous ways. It is up to the implementer of the programming tool to consider if and how such connections should be prevented.

Fig. 10 below shows practical examples of how stacking or crossing the lines in a particular way can visually obfuscate the logic.

V. Practical Evaluation

I evaluated the obfuscation techniques listed in Section IV by experimenting with five different freely available PLC programming environments:

1) CODESYS[5]
2) TwinCAT 3[6] by Beckhoff
3) Connected Components Workbench (CCW)[7] by Rockwell Automation
4) PC WORX[8] by Phoenix Contact
5) GEB Automation[9]

TwinCAT 3 seems to be based on CODESYS, and the observations for those two tools are the same.

A. Non-Latin characters in identifiers

Every tested tool prevented the user from using Greek, Cyrillic or Roman Numeral Unicode characters in identifiers. Still, we can try a simpler method of replacing uppercase "O" with zero, or uppercase "I" with lowercase "L". However, with the exception of GEB Automation, the tools use different visualisations which prevent the reader from confusing a user-specified simulacra block with a standard function block.

As seen in Fig. 9:

- In CODESYS (and therefore TwinCAT), many standard blocks like TON have a graphical icon inside the block element. The reader will differentiate between TON and our custom "T0N" (with a zero).
- In CCW and PC WORX, the user-specified blocks have a different color, and an automatically inserted instance name. The reader will spot that our custom "SlN" and "DlV" (both with lower-case "L") are not actually SIN and DIV.

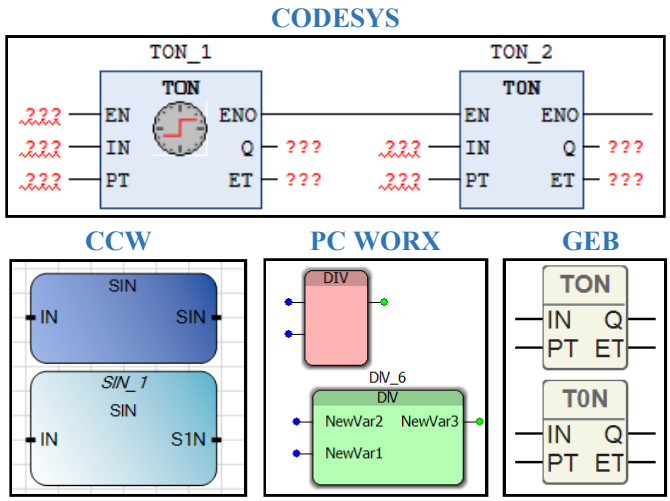

Figure 9. With the exception of GEB Automation, user-specified blocks are clearly distinguishable from standard blocks.

B. Logic hidden in execution control parameters

This technique is possible in every tool except CCW, which only supports the EN/ENO variables in LD programming.

(Although a standard MUX with just one IN input is unfortunately not supported in any of the tools. In CCW, the only options are four or eight IN inputs.)

C. Garbage insertion

The technique is universally applicable to all FBD engineering tools.

D. Masking in the diagram

All the tested tools prevented the user from placing a function block on top of another block.

E. Crossing signal lines

In CODESYS (and therefore TwinCAT), the blocks and their connection lines are placed automatically on the diagram, and cannot be moved by the programmer.

In CCW, the connection lines have curved angles, and a line crossing is made explicit with a small jump curve automatically added to the intersection point. However, it is still possible to manually position the lines so that the connections become visually ambiguous (see Fig. 10).

In PC WORX, the connection lines are drawn automatically, and their overlapping is not possible. A small arrow appears

[5]https://www.codesys.com/
[6]https://www.beckhoff.com/en-us/products/automation/twincat/
[7]https://www.rockwellautomation.com/en-us/capabilities/industrial-automation-control/design-and-configuration-software.html
[8]https://www.phoenixcontact.com/en-us/products/programming-pc-worx-express-2988670
[9]https://www.gebautomation.com/

in lines where data is flowing upwards, further preventing obfuscation attempts (see Fig. 10).

GEB Automation allows the user to manually and very flexibly wire the block connections in potentially confusing ways (see Fig. 10).

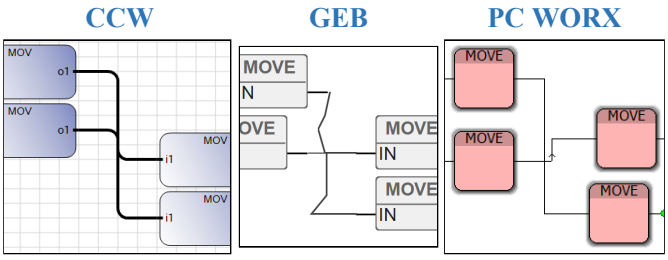

Figure 10. Ambiguous connection lines are possible in CCW and GEB Automation, but not in PC WORX.

VI. Discussion

A. The applicability of the techniques

The practical evaluation was limited to five PLC programming tools, of which two proved to be practically identical. Still, this set was sufficient for demonstrating that:

1) Different programming tools have differing, mostly successful mechanisms in place to prevent the programmer from deliberately introducing ambiguity.
2) Save for masking, every technique proved applicable in at least one tool.

The only caveat of the limited set of evaluated tools is that results do not indicate if the non-Latin character or masking techniques are applicable in any available tool. As a substitute for the non-Latin character method, the zero-for-O or l-for-1 replacement is still feasible in, at least, GEB Automation.

Garbage insertion is the most generally applicable method, and also the only method listed here that is both well-known and not specific to FBD or a similar visual programming language.

Of the tools, GEB Automation allowed us to apply the most techniques.

B. Formal verification as a supportive tool

An engineer purposefully obfuscating a program can use formal verification (by means of model checking) to prove that the transformed logic shall always exhibit the intended behaviour. For a simple example, let us verify that the patterns in Section IV-C work as we intend.

For each pattern, we can use Linear temporal logic (LTL) [20] to state: $\mathbf{G}$(BOOL_IN $\leftrightarrow$ BOOL_OUT) (i.e., the output shall always have a value equivalent with the "actual" input). For the logic in Fig. 7, we can also specify: $\mathbf{G}\,\neg$TON.Q (i.e., the output of the TON shall always be FALSE). Similarly, for the logic in Fig. 8: $\mathbf{G}\,\neg$AND.OUT.

We can then use a *model checker* like NuSMV [22] to prove those properties for a model of the logic.

(Alternatively, we can try and prove the Computation tree logic (CTL) [20] property: $\mathbf{AG}\,\mathbf{EF}$ TON.Q ("always-globally exists-finally" a state where TON.Q is TRUE), which will result in a counterexample because TON.Q indeed can never be TRUE. (See also [19].))

The exemplar NuSMV input files are found in the annex.

C. Further work

The garbage insertion technique requires the programmer to come up with superfluous logic built to just confuse a reader. This technique could benefit from code synthesis, where the FBD program is automatically generated based on, e.g., formal property specifications [23]. A range of techniques for FBD synthesis is discussed in [24].

The concern with synthesis is that automatically generated block diagrams are not necessarily acceptable as such [23] or easy to understand [24]. From our point of view, this is not a limitation but a benefit. Accordingly, a topic for future work would be the automatic generation of a FBD program as confusing as possible.

Motivated by [2], the applicability of abusing conditional and conditional jumps in FBD programs might warrant further study.

As Lopes points out in [13], the automatic translation of a program from, e.g., LD to SFC can at least partially obfuscate the logic. Accordingly, a tool-supported approach where the logic is translated many times between several languages (and then finally back to FBD) could assist in obfuscation.

VII. Conclusion

Stuxnet [4] and other successful attacks have shown that isolation from the Internet and use of dedicated devices are insufficient means to protect critical industrial systems from insertion of malware that can modify PLC code. Protection against novel cybersecurity threats is challenging due to the sheer number of bad actors striving to find every obscure means of attack. (Think the data on your computer is safe because you are not connected to any network? How about a piece of malicious code regulating the rotation speed the chassis fan to encode data in the sound heard over air? [25]).

Accordingly, the obfuscation of FBD programs is not a topic to dismiss, even (1) if the FBD programming tools can prevent the most obvious techniques, (2) rigorous verification will reveal unwanted program behaviour, dead code, etc., and (3) the whole topic might seem far-fetched upon first thought.

On a more positive note, obfuscation techniques are also worth exploring due their usability in protecting the intellectual property of I&C software engineers and companies, or to prevent any attacker from understanding a program they have managed to access without authority.

My work has shown that (1) developers of FBD programming tools need to pay attention to preventing the abuse of the language, and (2) there could be a need to clarify the rules in the next edition of IEC 61131-3 to prevent such abuse.

Acknowledgment

The ideas behind Section IV-B were partially inspired by Tom Murphy VII's work on logic gates [26]. The patterns

in Section IV-C are not my own invention, so I would like to credit the I&C engineers who came up with them unintentionally.

REFERENCES

[1] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, 2002.

[2] K. Yang, X. Lin, and L. Sun, "CShield: Enabling code privacy for cyber–physical systems," *Future Gener. Comput. Syst.*, vol. 125, pp. 564–574, 2021.

[3] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, "Diversification and obfuscation techniques for software security: A systematic literature review," *Inf. Softw. Technol.*, vol. 104, pp. 72–93, 2018.

[4] T. M. Chen and S. Abu-Nimeh, "Lessons from Stuxnet," *Computer*, vol. 44, no. 4, pp. 91–93, 2011.

[5] A. Di Pinto, Y. Dragoni, and A. Carcano, "TRITON: The first ICS cyber attack on safety instrument systems," in *Proc. Black Hat USA*, Aug. 2018.

[6] IEC, "Programmable controllers – part 3: Programming languages, Ed. 3.0," International Electrotechnical Commission, IEC Standard 61131-3:2013, 2013.

[7] S. Banescu and A. Pretschner, "Chapter five - a tutorial on software obfuscation," ser. Adv. Comput., A. M. Memon, Ed., 2018, vol. 108, pp. 283–353.

[8] A. Pakonen, I. Buzhinsky, and K. Björkman, "Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems," *Reliab. Eng. Syst.*, vol. 205, p. 107237, 2021.

[9] PLCopen. (2023) Status IEC 61131-3 standard. [Online]. Available: https://plcopen.org/status-iec-61131-3-standard

[10] N. Zubair, A. Ayub, H. Yoo, and I. Ahmed, "Control logic obfuscation attack in industrial control systems," in *Proc. CSR*, July 2022, pp. 227–232.

[11] Y. Ishigaki, N. Fujieda, Y. Matsuoka, K. Uyama, and S. Ichikawa, "An obfuscated hardwired sequence control system generated by high level synthesis," in *Proc. CANDAR*, Nov. 2017, pp. 323–325.

[12] M. Schwartz, J. Mulder, A. R. Chavez, and B. A. Allan, "Emerging techniques for field device security," *IEEE Secur. Priv.*, vol. 12, no. 6, pp. 24–31, 2014.

[13] V. Lopes, "Converting LD to SFC (IEC 61131-3)," Ph.D. dissertation, Univ. Porto, July 2017. [Online]. Available: https://hdl.handle.net/10216/105445

[14] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin, "IP protection and supply chain security through logic obfuscation: A systematic overview," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 6, Nov. 2019.

[15] K. Zamiri Azar, H. M. Kamali, S. Roshanisefat, H. Homayoun, C. P. Sotiriou, and A. Sasan, "Data flow obfuscation: A new paradigm for obfuscating circuits," *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, vol. 29, no. 4, pp. 643–656, 2021.

[16] ISO,IEC, "Information technology – Universal coded character set (UCS)," International Organization for Standardization / International Electrotechnical Commission, ISO/IEC Standard 10646:2020, 2020.

[17] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principle*. The MIT Press, 2015.

[18] Areva NP. (2012) U.S. EPR Protection System, Technical Report ANP-10309NP, Revision 4. [Online]. Available: https://www.nrc.gov/docs/ML1216/ML121660317.html

[19] A. Pakonen, "Oops! Examples of I&C design issues detected with model checking," in *Proc. ISOFIC*, November 2021. [Online]. Available: https://cris.vtt.fi/files/53941549/Pakonen_ISOFIC_2021_.pdf

[20] E. Clarke, O. Grumber, and D. Peled, *Model checking*, 2nd ed. Cambridge, Massachusetts, US: MIT press, 2001.

[21] I. Buzhinsky and A. Pakonen, "Symmetry breaking in model checking of fault-tolerant nuclear instrumentation and control systems," *IEEE Access*, vol. 8, pp. 197 684–197 694, 2020.

[22] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An opensource tool for symbolic model checking," in *International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Springer, 2002.

[23] J. Yoo, S. Cha, C. H. Kim, and D. Y. Song, "Synthesis of FBD-based PLC design from NuSCR formal specification," *Reliab. Eng. Syst.*, vol. 87, no. 2, pp. 287–294, 2005.

[24] M. Weiß, P. Marks, B. Maschler, D. White, P. Kesseli, and M. Weyrich, "Towards establishing formal verification and inductive code synthesis in the PLC domain," in *Proc. INDIN*, July 2021, pp. 1–8.

[25] M. Guri, Y. Solewicz, and Y. Elovici, "Fansmitter: Acoustic data exfiltration from air-gapped computers via fans noise," *Computers & Security*, vol. 91, p. 101721, 2020.

[26] T. Murphy, VII, "NaN gates and flip FLOPS," in *Proc. SIGBOVIK*, April 2019, pp. 98–102.