Code Diversification Mechanisms for Securing the Internet of Things [*]

Shukun Tokas, Olaf Owe, and Christian Johansen

University of Oslo, Norway

Internet of Things (IoT) is the networking of physical objects (or things) having embedded various forms of electronics, software, and sensors, and equipped with connectivity to enable the exchange of information. IoT is gaining popularity due to the great benefits it can offer in domestic and industrial settings as well as public infrastructures. However, securing IoT systems has proven a complex task, which is largely disregarded by industry for which the business driving force asks for functionality instead of safety and security. Securing IoT is also made difficult because of the resource constraints on the majority of these devices, which need to be cheap. Moreover, IoT devices are meant to be deployed in large numbers.

The fact that such a large amount of devices are programmed in the same way allows an attacker to exploit one vulnerability in millions of devices at once, thus with much more gains at the same cost. To address this challenge we propose to consider inclusion of diversification and randomisation mechanisms, at program design, implementation, and execution levels of IoT systems, to diversify observable program behaviour and thus increase resilience. By resilience we mean the ability to resist against attacks and the ability to recover quickly and with limited damages in case of infringements. Although diversity cannot protect against all kinds of attacks, it has proven a strong defence mechanism.

The idea of software diversity dates back to mid nineties, and has continued as research topic. Several comprehensive surveys and techniques made recently are [1, 2, 3]. Diversity techniques can be simply summarized as introducing uncertainty in the targeted program. Detailed knowledge of the target software (i.e., the exact binary rather than the high level code) is essential for a wide range of attacks, like memory corruption attacks, including control injection [4, 5, 6, 7]. This makes diversity a general defence mechanism that offers protection, providing "security by obscurity". Diversity techniques strive to include in software implementations high entropy so the attacker has a hard time figuring out the exact internal functioning and control of the system. The range of techniques for diversification through program transformation is large, and include approaches that vary with respect to threat models, security, performance, and practicality [8]. Software diversification has been applied at all levels of software, reaching the microprocessors level [9], the compiler [10] or the network [11].

We are interested in automated diversification techniques, in particular, techniques that can be employed at design and compile time. Such techniques could be deployed e.g., on version servers that distribute updates or patches to upgrade IoT devices in a seamless manner. One example of a manual diversification technique that one could think of automating is the software design methodology *N-variant* [12]. The need for N teams of developers developing N variants of the same software independently, from a common specification, should be replaced with automated techniques based on algorithms with mathematical guarantees (e.g., probabilistic or logical guarantees) that would produce the N variants from the same software specification, or implementation given by only one team of developers (e.g., [13]).

Automated techniques from programming languages like information flow static analysis [14] have been extended to the dynamic setting to protect against code injection. Dynamic taint

[*]This work was partially supported by the projects IoTSec and Diverse IoT.

analysis [15] automatically detects injection attacks without need for source code or special compilation for the monitored program, and hence works on commodity software. TaintCheck [15] was an example tool that can perform dynamic taint analysis by performing binary rewriting at run time. The technique was shown useful against cross-site scripting attacks [16]. Such techniques are still very popular and have been e.g., adopted for mobile operating systems [17] to protect the privacy of mobile apps [18]. Automated software diversification can also be used to counter bugs in software at runtime, thus making the system more robust, and applications to embedded systems have been proposed [19].

However, the diversification techniques are usually developed for standard operating systems or processor architectures running on powerful computing devices like PCs or phones. There is very little research on which mechanisms can be applied to IoT and how. In consequence, we take a particular interest in diversification techniques that are applicable in IoT and their programming domain, s.a.: program obfuscation, insertion of non-functional code, or function outlining. *Program Obfuscation* can be used for generating software variants since it transforms a source program P into a (functionally) equivalent program P' [20], and the program is obfuscated in such a way that it is difficult (not impossible) to reverse engineer (see also patent [21]). For example, variables and method names can be renamed or local variable names can be removed to make it difficult for the attacker to extract the values. A code obfuscater contains several components: preprocessor, intermediate code constructor, random code constructor etc., each component adding some form of obfuscation to the code. Obfuscation can be applied to data as well to prevent attack based on reverse engineering and code tampering. *Non-functional code* can be inserted to generate delay in execution or to indicate some space reservation in program memory. For example, when adding a No Operation Performed (NOP) instruction it consumes only one clock cycle because it does not affect any register. It can also be used to detect control flow change due to instruction misalignment. Another IoT relevant technique is *function Outlining*, in which a block is extracted from a function and then encapsulated in its own subroutine [1]. For example, a function may be split into two and all the local variables up to the point of split are passed as parameters to second function. This technique randomises the number of function calls, content of function, and code content. It protects systems from attacks based on code matching.

We plan to adapt, implement, and test the above techniques for IoT systems, and to analyse how they can be combined. At a higher abstraction level, we want to propose and implement a new techniques where we want to make use of modern concurrent programming languages like Creol [22] for developing the IoT system. We then take advantage of the inherent *non-determinism* of concurrent programs to produce numerous sequentialized versions based on varied thread scheduling policies (involving randomness). These sequential programs are the ones deployed on the actual IoT device, preferably also going through more transformations as above. This technique would prevent attacks based on knowledge of the precise timing of events.

References

[1] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “Sok: Automated software diversity,” in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 276–291, IEEE, 2014.

[2] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, “A survey on aims and environments of diversification and obfuscation in software security,” in *17th Int. Conf. Computer Systems and Technologies*, CompSysTech, pp. 113–120, ACM, 2016.

[3] B. Baudry and M. Monperrus, “The multiple facets of software diversity: Recent developments in year 2000 and beyond,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack,” in *6th ASIACCS Symposium*, pp. 30–40, ACM, 2011.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to risc,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS ’08, pp. 27–38, ACM, 2008.

[6] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” in *2012 IEEE Symposium on Security and Privacy*, pp. 601–615, IEEE, May 2012.

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 2:1–2:34, Mar. 2012.

[8] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “Sok: Automated software diversity,” in *2014 IEEE Symposium on Security and Privacy*, pp. 276–291, IEEE, May 2014.

[9] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: Accommodating software diversity in chip multiprocessors,” in *34th ISCA Symposium*, pp. 186–197, ACM, 2007.

[10] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, *Compiler-Generated Software Diversity*, pp. 77–98. Springer, 2011.

[11] A. J. O’Donnell and H. Sethu, “On achieving software diversity for improved network security using distributed coloring algorithms,” in *11th CCS Conference*, pp. 121–131, ACM, 2004.

[12] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[13] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, “N-variant systems: A secretless framework for security through diversity.,” in *USENIX Security Symposium*, pp. 105–120, 2006.

[14] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, Jan 2003.

[15] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, The Internet Society, 2005.

[16] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, “Cross site scripting prevention with dynamic data tainting and static analysis,” in *Network and Distributed System Security Symposium, NDSS*, The Internet Society, 2007.

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Trans. Comput. Syst.*, vol. 32, pp. 5:1–5:29, June 2014.

[18] M. L. Polla, F. Martinelli, and D. Sgandurra, “A survey on security for mobile devices,” *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[19] A. Höller, T. Rauter, J. Iber, and C. Kreiner, “Towards dynamic software diversity for resilient redundant embedded systems,” in *Software Eng. for Resilient Systems*, pp. 16–30, Springer, 2015.

[20] C. S. Collberg and C. D. Thomborson, “Watermarking, tamper-proofing, and obfuscation - tools for software protection,” *IEEE Transact. Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[21] C. Collberg, C. Thomborson, and D. Low, “Obfuscation techniques for enhancing software security,” Dec. 23 2003. US Patent 6,668,325.

[22] E. B. Johnsen, O. Owe, and I. C. Yu, “Creol: A type-safe object-oriented model for distributed concurrent systems,” *Theoretical Computer Science*, vol. 365, no. 1-2, pp. 23–66, 2006.