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 has proven a complex task, which is largely disregarded by industry for which the business driving force asks for functionality instead of safety or security. Securing IoT is also made difficult by of the resource constraints on the majority of these devices, which also need to be cheap.

IoT devices are often 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.

Software diversity is a research topic with several recent comprehensive surveys [1, 2]. 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 [3, 4, 5]. Diversity techniques strive to include in software implementations high entropy so the attacker has a hard time figuring out the exact internal functioning 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 [1].

Software diversification has been applied at all levels of software, reaching the microprocessors level, the compiler or the network. Automated techniques from programming languages like information flow static analysis [6] have been extended to the dynamic setting to protect against code injection. Dynamic taint analysis [7] automatically detects injection attacks without need for source code or special compilation for the monitored program, and hence works on commodity software. TaintCheck [7] was an example tool that performs binary rewriting at run time. Such techniques are still very popular and have been e.g., adopted for mobile operating systems [8] to protect the privacy of mobile apps [9]. It is interesting to see how such modern dynamic analysis techniques can be coupled with diversification techniques. 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 [10].

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 diversification mechanisms can be applied to IoT and how. Moreover, we are interested in automated diversification techniques, in particular, techniques that can be employed at design and compile time, because these could be deployed e.g., on

*This work was partially supported by the projects IoTSec and DiversIoT.

version servers that distribute updates or patches to upgrade IoT devices in a seamless manner. When trying to apply a diversification to IoT we are faced with two challenges: (I) IoT devices are resource constrained (with limited computational and memory capabilities), and (II) we need to generate a significant number of software variants (due to large number of IoT devices).

Following are some of the relevant techniques:

N-variant Technique One example of a manual diversification technique that one could think of automating is the software design methodology *N-variant* [11]. 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., [12]).

Overhead: Does not have any execution overhead, thus being good for the resource constraint nature of IoT devices. However, the overhead is in terms of programming resources (budget, skills, time) required during development of the variants. It moreover incurs an overhead proportional to N for maintenance and updates.

Applicability: This mechanism seems to be useful when it employs 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. This mechanism is useful for developing a fault tolerant system, as diverse sources of faults leads to transient effects.

Program Obfuscation Code transformation techniques change the source program P into a (functionally) equivalent program P' [13]. The objective is to make low-level semantics of programs harder and more complex for attacker to comprehend, without affecting the program's observable behavior. However, to have effective security and diversity the obfuscated code should be difficult enough to reverse engineer. Collberg et al [13], identified four main classes of transformation for code and data obfuscation: lexical transformation, control flow transformation, data flow transformation, and preventive transformation. These may involve renaming variable, altering control flow of program by using opaque predicates or graph flattening, changing the data encoding, etc. After applying a series of transformations, the obfuscated code is distributed to clients. This technique is an effective defence against attacks based on reverse engineering and code tampering.

Overhead: However, it does incur an added cost due to memory usage and execution cycles required to execute obfuscated code.

Applicability: Benefit of this technique is that it can be automated to generate large number of code variants, in a platform independent manner (considering transformation at source code level). It diversifies the code in terms of code space and execution timings, and also it is effective against automated program analysis.

Insertion of Non-Functional Code Non-functional code can be inserted to generate delay in execution or to indicate some space reservation in program memory. Adding any number of NOP instruction does not change program semantics, but it generates diverse binaries and makes the program execution more unpredictable to the attackers as the variants will have different execution statistics. It can also be used to detect control flow change due to instruction misalignment.

Overhead: Consumes only one clock cycle, overhead is proportional to number of NOP instructions included.

Applicabiity: It doesn't degrade systems performance significantly and can be combined with other diversification mechanisms to have an effective diversification strategy.

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 technique where we want to make use of modern concurrent programming languages like Creol [14] 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 will be 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. We plan to develop and demonstrate this idea in detail using a case study.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.