Mismorphism: The Heart of the Weird Machine

Prashant Anantharaman[1], Vijay Kothari[1], J. Peter Brady[1], Ira Ray Jenkins[1], Sameed Ali[1], Michael C. Millian[1], Ross Koppel[3], Jim Blythe[2], Sergey Bratus[1], and Sean W. Smith[1]

[1] Dartmouth College, Hanover, NH, USA
[2] Information Sciences Institute, University of Southern California, Los Angeles, CA, USA
[3] Sociology Department, University of Pennsylvania, Philadelphia, PA, USA

Abstract. Mismorphisms—instances where predicates take on different truth values across different interpretations of reality (notably, different actors' perceptions of reality and the actual reality)—are the source of weird instructions. These weird instructions are tiny code snippets or gadgets that present the exploit programmer with unintended computational capabilities. Collectively, they constitute the weird machine upon which the exploit program runs. That is, a protocol or parser vulnerability is evidence of a weird machine, which, in turn, is evidence of an underlying mismorphism. This paper seeks to address vulnerabilities at the mismorphism layer.

The work presented here connects to our prior work: Language-theoretic security, LangSec, provides a methodology for eliminating weird machines: By limiting the expressiveness of the input language, separating and constraining the parser code from the execution code, and ensuring only valid input makes its way to the execution code, entire classes of vulnerabilities can be avoided. Here, we go a layer deeper with our investigation of the mismorphisms responsible for weird machines.

In this paper, we re-introduce LangSec and mismorphisms, and we develop a logical representation of mismorphisms that complements our previous semiotic-triad-based representation. Additionally, we develop a preliminary set of classes for expressing LangSec mismorphisms, and we use this mismorphism-based scheme to classify a corpus of LangSec vulnerabilities.

1 Introduction

(

- need to stress that we don't use the power of the predicates much in this paper... we're mostly dealing with propositional variables.
- clarify definition of predicate if needed
- we might want to present situations where an interpretor interprets a predicate with incomplete information as one in which variables are not assigned or propositional variables cannot be evaluated (rather than introducing U as

a value that all variables can take on, as opposed to the final interpretation value)... or we should represent U as being a fixed value that is unknown to the interpretor.

- vk)

Mismatches between the perceptions of the designer, the implementor, and the user often result in protocol vulnerabilities. The designer has a high-level vision for how they believe the protocol should function, and this vision guides the creation of the specification. In practice, the specification may diverge from the initial vision due to practical constraints, e.g., hardware or real-time requirements. The implementor then produces code to meet the specification based on their perceptions of how the protocol should function and, in some cases, how the user will interact with it. However, incorrect assumptions may produce vulnerabilities in the form of bugs or unintended operation. A user—informed by their own assumptions and perceptions—may then interact with a system or service that relies upon the protocol. A misunderstanding of the protocol and its operation can drive the user toward a decision that produces an unintended outcome. Ultimately, the security of the protocol rests on the consistency between the various actors' mental models of the protocol, the protocol specification, and the protocol implementation.

A *mismorphism* refers to a mapping between different representations of reality (e.g., the distinct mental model of the protocol designer, the protocol implementor, and the end user) for which properties that ought to be preserved are not. In the past, we have used this concept and an accompanying semiotic-triad-based model to succinctly express the root causes of usable security failures [18]. We now apply this model to protocol design, development, and use. As mentioned earlier, many vulnerabilities stem from a mismatch between different actors' representations of protocols and the protocol operation in practice, e.g., the HeartBleed [9] vulnerability embodies a mismatch between the protocol specification, which involved validating a length field, and the implementation, which failed to do so. Therefore, it is natural to adopt the mismorphism model to examine the root causes of protocol vulnerabilities. That is precisely what we do in this paper.

The notion of mismorphism closely parallels the views expressed by Bratus et al. [5] in their discussion of exploit programming:

> "Successful exploitation is always evidence of someone's incorrect assumptions about the computational nature of the system— in hindsight, which is 20-20."

The mindset embodied in this quote forms the foundation for the field of language-theoretic security, LangSec.[4] Exploitation is unintended computation

[4] We only give a brief primer of LangSec in this paper. For those who are interested in learning more we recommend consulting the LangSec website [4] located at `http://langsec.org`.

performed on a *weird machine* [17] that the target program harbors. [5] Weird machines comprise the gadgets within a program that offer the adversary unintended computational capabilities to carry out an attack, e.g., NOP sleds, buffer overflows. Weird machines were never intended by protocol designers or implementors but organically arose within the protocol design and development phases. For instance, consider a designer who intends to design a web server program. The implementor of the software attempts to sanitize the input, but unwittingly lets malicious input through. This enables the adversary to supply unexpected input, resulting in unexpected behavior. The adversary repeatedly observes instances of unspecified program behaviour and uses these observations to craft an exploit program that they run on the exposed weird machine; this weird machine may serve as a Turing machine or otherwise expressive machine for the supplied input, the exploit program. To the designer, it was just a web server program; to the exploit programmer it provides an avenue of attack.

LangSec advocates taking a principled approach— one that is informed language theory, automata theory, and computability theory— to parser design and development to eliminate the weird machine. LangSec facilitates the construction of safer protocols that behave closer to what the way that designers and implementors envisioned by identifying best practices for protocol construction, such as parsing the input in full before program execution and ensuring the parser obeys known computability boundaries for safer computation. It also delivers tools such as Hammer and McHammer to achieve these goals [13,15].

Momot et al. [14] created a taxonomy of LangSec anti-patterns and used it to suggest ways to improve the Common Weakness Enumeration (CWE) database. Erik Poll [16] takes a different approach, classifying input vulnerabilities into two broad categories — flows in processing input and flaws in forwarding input— and discusses examples from both categories in detail.

We build upon this prior research, viewing mismorphisms as precursors to the weird machine. Our work is motivated by the belief that identifying and categorizing the mismorphisms that produce weird machines is a valuable step in systematically unpacking the causes of vulnerabilities and ultimately addressing them.

2 Mismorphisms

Here, we provide a very brief primer on mismorphisms and present a logical model for capturing them. Our work here builds upon our earlier efforts to build a semiotic-triad-based mismorphism model [18], which was, in turn, inspired by

[5] For the reader interested in learing more about weird machines: Dullien [8] provides a formal definition for understanding weird machines and shows that it is feasible to build software that is resilient to memory corruption. Bratus and Shubina [6] also presents exploit programming as a problem of code reuse, discusses how the adversary uses code presented by the weird machine to carry out the exploit, and describes colliding actors' abstractions of how the code works.

early semiotics work by pioneers Ogden and Richards [7]. The logical representation presented here blends temporal logic with the notion of interpretation. Following this section, we demonstrate how this logical model can be used to classify underlying causes of LangSec vulnerabilities by providing a preliminary classification using real-world examples.

In the context of this paper, a mismorphism refers to a difference in interpretations between two or more interpretors. That is, we can think of different interpretors (e.g., people) interpreting propositions or predicates about the world. In general, it is good when the interpretations agree and are in accordance with reality. However, when a predicate takes on a truth value under one interpretation but not another interpretation, we have a mismorphism, which may be a cause for concern. In our earlier applications, we found these mismorphisms were useful in understanding usable security failures and user circumvention. Here, we apply them to protocol and parser security. As mismorphisms deal with interpretations of predicates and how interpretations differ between interpretors, it's easy to see why formal logic provides a natural foundation to represent them.

We use the words *predicate* and *interpretation* in similar—albeit, not identical—manners to their formal-logic meanings, e.g., as presented by Aho and Ullman [3]. We refer to a predicate as a function from variables to values that belong to some set $R \cup \{U\}$. Here, R is a set of values and U is a special value that means uncertain; we will soon explain the need for U, but for now, one can think of U as an unassigned variable. We refer to an *interpretation* of a predicate as an assignment of values to variables, which results in the predicate being interpreted as T (true), F (false), or U (uncertain/unknown). A predicate is interpreted as T if after substituting all variables for their truth values, every possible substitution of U values for values in R results in the predicate being interpreted as T; it is interpreted as F if after substituting all variables for their truth values, every possible substitution of U values for values in R results in the predicate being interpreted as F; if the interpretation is neither T nor F, it is interpreted as U.

The interpretation must, of course, be done by someone (or perhaps something) and that someone is the *interpretor*. In this paper, common *interpretors* include the oracle O who interprets the predicate as it is in reality, the designer D, the implementor I, and the user U. We note that some interpretors may be people who do not feel they have adequate information to assign values to variables or to determine the truth value of a predicate. It is imperative we allow for the uncertain value U to capture such instances.

To represent mismorphisms we need a way to express the relationship between interpretations of a predicate. Thus, we have the following:

$$[Predicate]\ [Interpretation\ Relation]\ [List\ of\ interpretors]$$

The interpretation relations over the set of all predicate-interpretor pairs are k-ary relations where $k >= 2$ is the number of interpretors there are in the interpretation relation— and each k-ary relation is over the interpretations of the predicate by the k interpretors. The three interpretation relations we are concerned with in this paper are: the interpretation-equivalence relation ($\underset{\text{interp.}}{=}$), the interpretation-uncertainty relation ($\underset{\text{interp.}}{\overset{?}{=}}$), and the interpretation-inequivalence relation ($\underset{\text{interp.}}{\neq}$).[6] These relations are defined as follows where each P represents a predicate and each A_i represents an interpretor:

- $P \underset{\text{interp.}}{=} A_1, A_2, \ldots A_k$ iff P interpreted by each A_i has a truth value that's either T or F (never U)— and all interpretations yield the same truth value.
- $P \underset{\text{interp.}}{\overset{?}{=}} A_1, A_2, \ldots A_k$ iff P takes on the value U when interpreted by at least one A_i.
- $P \underset{\text{interp.}}{\neq} A_1, A_2, \ldots A_k$ iff P interpreted by A_i is T and P interpreted by A_j is F for some $i \neq j$.

There are a few important observations to note here. One is that the oracle O will always hold the correct truth value for the predicate by definition. Another is that if we only know the $\underset{\text{interp.}}{\overset{?}{=}}$ relation applies, we won't know which interpretor is uncertain about the predicate or even how many interpretor are uncertain unless $k = 2$ and one interpretor is the oracle. Similarly, if we only know that the $\underset{\text{interp.}}{\neq}$ relation applies, we do not know where the mismatch exists unless $k = 2$. That said, knowledge that the oracle O always holds the correct interpretation combined with other facts can help specify where the uncertainty or inequivalence stem from. Last, the $\underset{\text{interp.}}{=}$ relation can not be applicable if either the $\underset{\text{interp.}}{\overset{?}{=}}$ or the $\underset{\text{interp.}}{\neq}$ interpretations are applicable; however, $P \underset{\text{interp.}}{\overset{?}{=}} A_1, \ldots A_k$ and $P \underset{\text{interp.}}{\neq} A_1, \ldots A_k$ can both be applicable simultaneously.

The purpose of creating this model was to allow us to capture mismorphisms—and it does. Mismorphisms correspond to instances where either the interpretation-uncertainty relation or interpretation-inequivalence relation apply.

For our purposes, we also consider some natural extensions to this logical formalism. In select cases, we may consider multiple interpretors of the same role. In these instances, we assign subscripts to distinguish roles, e.g., D, I_1, I_2, O. Also, there are temporal aspects that may be relevant. Predicates can be functions of time and so can the interpretations. While we use the common $v(t)$-style notation to represent a variable as a function of time within a predicate, in select cases we consider the interpretor as a function of time, e.g., $I_4^{t_3}$ means the interpretation is done by implementor I_4 at time $t = t_3$.

[6] Note that for $k = 2$, if we confine ourselves to predicates that take on only T or F values, the relation $\underset{\text{interp.}}{=}$ is an equivalence relation in the mathematical sense, as one might expect, i.e., it obeys reflexivity, commutativity, and transitivity.

3 Preliminary Classification

We describe the various types of mismorphisms using a mathematical notation. All the vulnerabilities we catalog fall into one of the categories of mismorphisms we describe below.

3.1 Language is Decidable

Any input language format needs to be decidable for the implementor to be able to parse and make sure that there are no corner cases when the program can enter unexpected states or fail to terminate. When the designer assumes language L is decidable (in the absence of proof that it is), the program may harbor the potential for unexpected computation.

For one example, the Ethereum platform uses a Turing-complete input language to enable its smart contracts. It is demonstrably more difficult to build a parser for such an input language. Such added complexity led to the Ethereum DAO disaster [20], in which all ethers were stolen, forcing the developers to perform a highly controversial hard fork. As a result, some developers built a decidable version of Solidity called vyper [10].

We define such a language mismorphism by the form:

$$L \text{ is decidable} \underset{\text{interp.}}{\overset{?}{=}} O, D \tag{1}$$

A diagrammatic representation of the above formalism can be found in Figure 1.

3.2 Shotgun Parsers

Shotgun parsers perform input data checking and handling interspersed with processing logic. Shotgun parsers do not perform full recognition before the data is processed. Hence, implementors may assume that a field x has the same value at time t and time $t + \delta$, but the processing logic may change the value of the field x in an input buffer B.

This mismorphism relation is seen below:

$$B(t) = B(t + \delta) \underset{\text{interp.}}{\neq} O, I \tag{2}$$

Implementors expect the buffer to be intact across time, but that is not observed to be the case. Shotgun parsing can cause mismorphisms in two distinct ways. First, when a partially validated input is treated as though it is fully validated. Implementor 1 performs shotgun parsing, and knows input is only partially validated. Implementor 2 works on execution, and assumes the input is fully validated by the time the code segment is executed. This type of a shotgun parser mismorphism can be represented as the following:

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2 \tag{3}$$

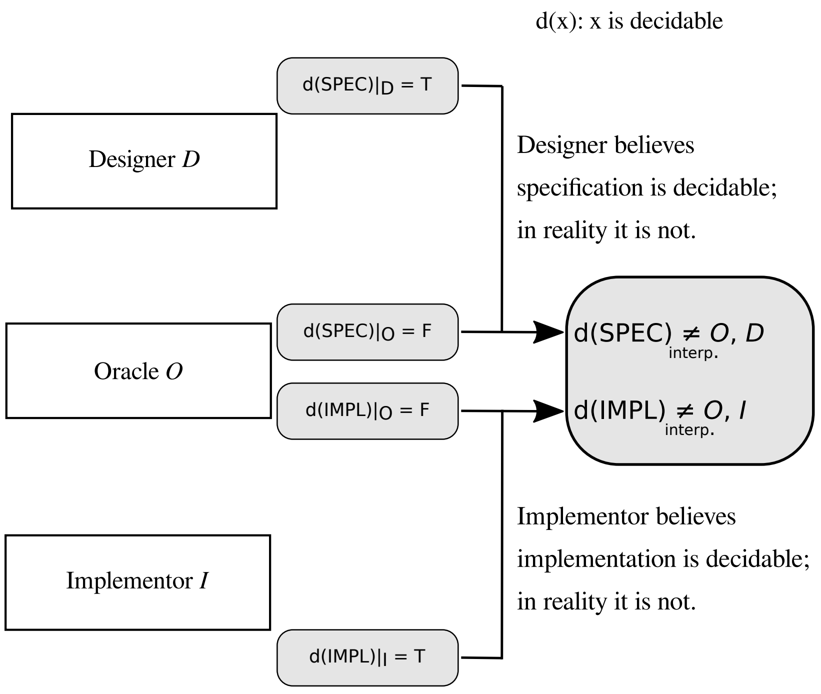
d(x): x is decidable
Designer D
Oracle O
Implementor I
d(SPEC)|D = T
d(SPEC)|O = F
d(IMPL)|O = F
d(IMPL)|I = T
Designer believes specification is decidable; in reality it is not.
Implementor believes implementation is decidable; in reality it is not.
$d(\text{SPEC}) \underset{\text{interp.}}{\neq} O, D$
$d(\text{IMPL}) \underset{\text{interp.}}{\neq} O, I$

Fig. 1. One class of mismorphism where implementors and designers both disagree with the reality that the language is actually undecidable.

Second, the same implementor performs shotgun parsing, and execution. But interprets the same protocol differently during different times. This type of a mismorphism can be represented as the following:

$$B \text{ is accepted } \underset{\text{interp.}}{\neq} I^{t_1}, I^{t_2} \tag{4}$$

3.3 Two parsers for same protocol aren't equivalent

Designers of protocols intend for two endpoints to have the exact same functionality, and build identical parse trees. The Android Master Key bug is an apt example for this type of a mismorphism [11]. The parsers for the unzipping function in Java and C++ were not equivalent, leading to a parsing differential.

We describe this relation as:

$$\text{Parsers } P_1, P_2 \text{ are equivalent } \underset{\text{interp.}}{\neq} O, D, I \tag{5}$$

3.4 Implementor is unaware that some fields need to be validated

Designers of protocols introduce new features in the specification of the protocol, sometimes not describing fully or accurately. The designer introduces a field x in the protocol, but the interpretor does not entirely understand how to interpret it. The Heartbleed vulnerability was an example of this. The designers included the heartbeat message, but the implementors did not completely understand it and missed an additional check to make sure the length fields matched [9].

$$\text{sanity check } C \text{ is performed } \underset{\text{interp.}}{\neq} O, D, I \tag{6}$$

3.5 Types of fields in the buffer is fixed for the life cycle of the buffer

The types of values that have already been parsed must remain constant. Sometimes, implementors assume field x is treated as type $t(x)$ throughout execution. in reality, field is treated as type $t(x)$ and $t'(x)$.

$$\mathit{type}(x) \text{ is fixed } \underset{\text{interp.}}{\neq} O, D, I \tag{7}$$

A summary of the vulnerabilities along with their mismorphisms is in Table 1.

Table 1: Summary of LangSec vulnerabilities and causes

Vulnerability	**Underlying Mismorphisms**
ShellShock	**Description:** Bash unintentionally executes commands that are concatenated to function definitions that are inside environment variables. **Mismorphism:** sanity check C is performed $\underset{\text{interp.}}{\neq}$ O, D, I The sanity check C here makes sure that once functions are terminated, the variable shouldn't be reading commands that follow it.
Rosetta Flash	**Description:** SWF files that are requested using JSONP are incorrectly parsed once they are compressed using *zlib*. Compressed SWF files can contain only alphanumeric characters [19]. **Mismorphism:** sanity check C is performed $\underset{\text{interp.}}{\neq}$ O, D, I The specification of the SWF file format is not exhaustively validated using a grammar. The fix uses conditions such as checking for the first and last bytes for special, non-alphanumeric characters.
HeartBleed	**Description:** The protocol involves two length fields, one that specifies the total length of the heartbeat message; the other specifies the size of the payload of the heartbeat message. **Mismorphism:** sanity check C is performed $\underset{\text{interp.}}{\neq}$ O, D, I Sanity check C involves verifying the length fields l_1 and l_2 match.

Android Master Key	**Description:** The Java and C++ implementations of the cryptographic library performing unzipping were not equivalent. **Mismorphism:** $\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I$
Ruby on Rails - Omakase	**Description:** The Rails YAML loader doesn't validate the input string and check that it is valid JSON. And it doesn't load the entire JSON; instead it just starts replacing characters to convert JSON to YAML [12]. **Mismorphism:** $\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$ Sanity check C should first recognize and make sure the JSON is well-formed, before replacing the characters in YAML.
Wireshark ASN.1 Bug	**Description:** The same integer length value was treated as `unsigned` by some parts of the code, and `signed` by others, leading to *weird* behavior. **Mismorphism:** $type(x) \text{ is fixed} \underset{\text{interp.}}{\neq} O, D, I$ The types of all the parsed fields must be fixed. The parse tree must be used as much as possible to process the parsed input.

Nginx HTTP Chunked Encoding	**Description:** Large chunk size for the Transfer-Encoding chunk size trigger integer signedness error and a stack-based buffer overflow [1]. **Mismorphism:** $B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2$ The shotgun parser works on execution without validating the value of the length field which could be much larger than allowed causing buffer overflows. All implementors must work with the same knowledge, and the input must first be recognized fully.
Elasticsearch Crafted Script Bug	**Description:** Elasticsearch runs Groovy scripts directly in a sandbox. Attackers were able to craft a script that would bypass the sandbox check and execute shell commands. **Mismorphism:** $L \text{ is decidable} \underset{\text{interp.}}{\overset{?}{=}} O, D$ Developers of Elasticsearch had to explore the option of abandoning Groovy in favor of a safe and less dynamic alternative.

Mozilla NSS Null Character Bug	**Description:** If domain names included a null character, there was a discrepancy between the way Certificate Authorities issued certificates and the way SSL clients handled them. Certificate Authorities issued certificates for the domain after the null character, whereas the SSL clients used the domain name ahead of the null character. **Mismorphism:** $\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I$ Although having a null character in a certificate is not accepted behavior, certificate authorities and clients do not want to ignore requests that contain them. So they follow their own interpretations, resulting in a parser differential.
Adobe Reader CVE-2013-2729	**Description:** In running length encoded bitmaps, Adobe Reader would write pixel values to arbitrary memory locations since there was a bounds check that was skipped. **Mismorphism:** $B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2$ The code used a shotgun parser where the implementor of the processing logic assumed all fields were validated. The bounds check was never performed.

OpenBSD - Fragmented ICMPv6 Packet Remote Execution	**Description:** Fragmented ICMP6 packets cause an overflow in the mbuf data structure in the kernel may cause a kernel panic or remote code execution depending on packet contents [2]. **Mismorphism:** $\text{sanity check } C \text{ is performed } \underset{\text{interp.}}{\neq} O, D, I$ Implementors of the ICMP6 packet structures in OpenBSD did not understand how to map it to the existing `mbuf` structure, and then validate it.

4 Conclusion

In this paper, we proposed a novel approach to categorizing the root causes of protocols vulnerabilities. We applied a semiotic-triad-based model of mismorphism, along with a newly developed logical model to create a preliminary set of mismorphism classes to understand those underlying LangSec vulnerabilities. We also created a corpus of vulnerabilities and used the classification scheme to classify the mismorphisms.

LangSec theorizes solutions to most input handling problems presented in this paper. LangSec offers parser-combinator toolkits and advocates for its use. Parser-combinator toolkits allow for efficient conversion of a protocol design specification into a parser. Unfortunately, this doesn't automatically translate to solutions for all the categories of mismorphisms we have encountered. In addition to an overview of the work presented in this paper, we aim to discuss tools that would help address and prevent mismorphisms created by designers and implementors of programs.

Acknowledgement

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0179 and Department of Energy under Award Number DE-OE0000780.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, DARPA, United States Government or any agency thereof.

References

1. CVE-2013-2028 Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow | Rapid7, https://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size
2. OpenBSD's IPv6 mbufs remote kernel buffer overflow, https://www.secureauth.com/labs/advisories/open-bsd-advisorie
3. Aho, A., Ullman, J.: Foundations of Computer Science: C Edition, Chapter 14 (July 1994), http://infolab.stanford.edu/~ullman/focs.html
4. Bratus, S.: LANGSEC: Language-theoretic Security: "The View from the Tower of Babel", http://langsec.org
5. Bratus, S., Locasto, M., Patterson, M., Sassaman, L., Shubina, A.: Exploit programming: From buffer overflows to weird machines and theory of computation. {USENIX; login:} (2011)
6. Bratus, S., Shubina, A.: Exploitation as code reuse: On the need of formalization. it-Information Technology **59**(2), 93–100 (2017)
7. C. Ogden and I. Richards: The Meaning of Meaning. In: Harcourt, Brace and Company (1927)
8. Dullien, T.F.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing (2017)
9. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al.: The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference. pp. 475–488. ACM (2014)
10. Ethereum: Pythonic Smart Contract Language for the EVM, https://github.com/ethereum/vyper
11. Freeman, J.: Exploit (& Fix) Android "Master Key", http://www.saurik.com/id/17
12. Helmkamp, B.: Rails' Remote Code Execution Vulnerability Explained, https://codeclimate.com/blog/rails-remote-code-execution-vulnerability-explained/
13. Hermerschmidt, L.: McHammerCoder: A binary capable parser and unparser generator, https://github.com/McHammerCoder
14. Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L.: The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In: Cybersecurity Development (SecDev), IEEE. pp. 45–52. IEEE (2016)
15. Patterson, M.: Parser combinators for binary formats, in C, https://github.com/UpstandingHackers/hammer
16. Poll, E.: Langsec revisited: input security flaws of the second kind. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 329–334. IEEE (2018)
17. Shapiro, R., Bratus, S., Smith, S.W.: " weird machines" in elf: A spotlight on the underappreciated metadata. In: WOOT (2013)
18. Smith, S.W., Koppel, R., Blythe, J., Kothari, V.: Mismorphism: a semiotic model of computer security circumvention. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. p. 25. ACM (2015)
19. Spagnuolo, M.: Abusing JSONP with Rosetta Flash, https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/
20. Torpey, K.: The DAO Disaster Illustrates Differing Philosophies in Bitcoin and Ethereum, https://www.coingecko.com/buzz/dao-disaster-differing-philosophies-bitcoin-ethereum