Self-Forensics Through Case Studies of Small-to-Medium Software Systems

Serguei A. Mokhov
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
Montreal, Quebec, Canada
Email: mokhov@cse.concordia.ca

Emil Vassev
Lero - The Irish Software Engineering Research Center
University College Dublin
Dublin, Ireland
Email: emil.vassev@lero.ie

Abstract

The notion and definition of self-forensics was introduced by Mokhov to encompass software and hardware capabilities for autonomic and other systems to record their own states, events, and others encoded in a forensic form suitable for (potentially automated) forensic analysis, evidence modeling and specification, and event reconstruction for various system components. For self-forensics, "self-dissection" is possible for analysis using a standard language and decision making if the system includes such a self-forensic subsystem. The self-forensic evidence is encoded in a cyberforensic investigation case and event reconstruction language, Forensic Lucid. The encoding of the stories depicted by the evidence comprise a context as a first-class value of a Forensic Lucid "program", after which an investigator models the case describing relationships between various events and pieces of information. It is important to get the context right for the case to have a meaning and the proper meaning computation, so we perform case studies of some small-to-medium, distributed and not, primarily academic open-source software systems. In this work, for the purpose of implementation of the small self-forensic modules for the data structures and event flow, we specify the requirements of what the context should be for those systems. The systems share in common the base programming language – Java, so our self-forensic logging of the Java data structures and events as Forensic Lucid context specification expressions is laid out ready for an investigator to examine and model the case.

Keywords

self-forensics; specification; Forensic Lucid; context-aware forensic computing; intensional programming; GIPSY; DMARF; JDSF; Cryptolysis

I. Introduction

The property of *self-forensics* [1], [2] is a new concept introduced to encompass and formally apply to or be included in the design of not only autonomic hardware and/or software systems, which are inherently complex and hard to analyze in general when incidents happen, but also as an optional requirement for smaller projects.

Self-forensics in a nutshell includes a dedicated module or modules observing the regular modules in some way, logging the observations and events that led to them in a predefined format suitable for automatic forensic processing and deduction, and event reconstruction in case of incidents. The modules can optionally have a capability of automatically diagnose themselves based on the collected evidence and make more complex and complete decisions after the analysis than ad-hoc binary decisions. In a sense, the self-forensic modules can be seen as smart "blackboxes" like in planes, but can be included in spacecraft, road vehicles, large and small software systems that assist with the incident analysis. Human experts can be also trained in investigation techniques based on the forensic data sets collected during different incidents. In a traditional sense, one argues that any self-diagnostics, as well as traditional logging, hardware or software, are necessary parts of self-forensics that have been around for long time.

We compile sample Forensic Lucid specifications for self-forensics of a few software projects as case studies. The specifications are there to be built into the system for the purpose of tracing and understanding complex relationships and events within some components of the said systems, especially the distributed ones. In this work, we reasonably narrowly focus on "exporting" the states of the systems as data structures in chronological encoded order as Forensic Lucid contexts using its syntax in accordance with the grammar for compiling the cases.

We further proceed to describe the background and the related work, followed by the specification of the core data structures and data flows in Forensic Lucid of the case studies, such as the Distributed Modular Audio Recognition Framework (DMARF), General Intensional Programming System (GIPSY), Java Data Security Framework (JDSF), and Cryptolysis – an automated cryptanalysis framework for classical ciphers. These are primarily academic/open-source systems with the first two being distributed and a lot more complex than the last two. After the specification, we conclude and list a few near future work items.

II. Background and Related Work

A. Forensic Computing

Forensic computing has traditionally been associated with computer crime investigations. Existing self-diagnostics, computer BIOS reports, and S.M.A.R.T. [3] reporting for hard disks as well as many other devices are a good source for such data computing, i.e. be more forensics-friendly and provide forensics interfaces for self-forensics analysis and investigation.

There is a wide scope of research done in forensic computing, including formal and practical approaches, methodologies, and the associated tools [4], [5], [6], [7], [8], [9]. A part of the related work on what we call *self-forensics* there have been some preliminary or related works done in the past, but not identified as such. For example, a state-tracing Linux kernel [10] exhibits some elements of self-forensics by tracing its own state for the purse of forensic analysis.

AspectJ [11]'s tracing capability of Java programs provides another way of when the forensic data of self can be collected during the forward tracing of the normal execution flow. Given a large number of web-based application deployed today are written in Java, the use of AspectJ, which isn't coupled with the applications' source code and can observe application objects independently, provides a good basis for a self-forensics middleware.

B. Forensic Lucid

Forensic Lucid [12], [13], [14] is a forensic case specification language for automatic deduction and event reconstruction of computer crime incidents. The language itself is general enough to specify any events, their properties, duration, as well as the context-aware system model.

Forensic Lucid is significantly influenced by and is meant to be a usable improvement of the work of Gladyshev et al. on formal forensic analysis and event reconstruction using finite state automate (FSA) to model incidents and reason about them [5], [4]. It has also some similarities with Focus [15]. Forensic Lucid is also based on Lucid [16], [17], [18], [19], [20] and its various dialects that allow natural expression of various phenomena, inherently parallel, and most importantly, context-aware, i.e. the notion of context is specified as a first-class value in Lucid [21], [22], [23]. All Lucid dialects are functional programming languages. All these properties make Forensic Lucid a good choice for self-forensic computing in self-managed systems to complement the existing self-CHOP properties (self-configuration, self-healing, self-optimization, and self-protection in autonomic computing [24], [25]).

For the purposes of portability or model-checking, depending on the evaluating engine, Forensic Lucid constructs comprising the context of the encoded self-forensic evidence can be translated into XML specifications (that can be unsatisfiable [26] and less compact, but maybe more portable) of a specification for embedded systems and the like or translated into the PRISM probabilistic model checker's language [27] for model validation.

Context: As was mentioned, Forensic Lucid is context-oriented where a crime scene model comprises a state machine of evaluation and the forensic evidence and witness accounts comprise the context. The basic context entities comprise an observation o in Equation 1, observation sequence os in Equation 2, and the evidential statement in Equation 3. These terms are inherited from the Gladyshev's work [5], [4] and in a nutshell represent the context of evaluation in Forensic Lucid. An observation of a property P has a duration between $[\min, \min + \max]$. An observation sequence os is a chronologically ordered collection of observations representing a story witnessed by someone or something. It may also encode a description of some evidence found. All these "stories" (observation sequences, or logs, if you will) together represent an evidential statement about an incident. The evidential statement es is an unordered collection of observation sequences. The property P itself can encode anything of interest – an element of any data type or even another Forensic Lucid expression, or an object instance hierarchy or an event.

$$o = (P, \min, \max) \tag{1}$$

$$os = \{o_1, \ldots, o_n\} \tag{2}$$

$$es = \{os_1, \ldots, os_m\} \tag{3}$$

Having constructed the context, one needs to built a *transition function* ψ and its inverse Ψ^{-1}. The generic versions of them are provided by Forensic Lucid [13] based on [4], [5], but the investigation-specific one has to be built, potentially visually using a data-flow graph (DFG) editor [28], by the investigators. The specific Ψ^{-1} takes evidential statement as an argument and the generic one takes the specific one.

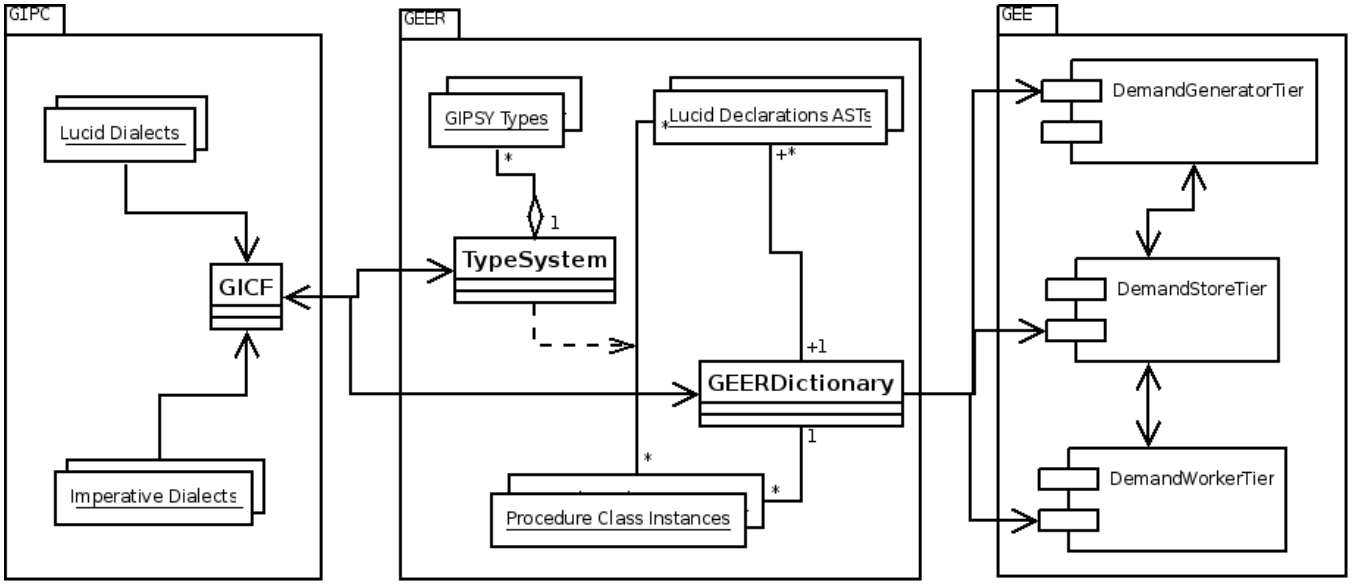

Figure 1. High-level structure of the GIPSY

C. General Intensional Programming System (GIPSY)

The General Intensional Programming System (GIPSY) [29], [30], [31], [32], [33], [34] is an open-source platform implemented primarily in Java to investigate properties of the Lucid [18], [19], [35] family of intensional programming languages and beyond. It is a distributed system, designed as a modular collection of frameworks where components related to the development (RIPE, a Run-time Integrated Programming Environment), compilation (GIPC, the General Intensional Programming Compiler), and execution (GEE, the General Eduction Engine) of Lucid programs are decoupled to allow easy extension, addition, and replacement of the components. GIPSY has a collection of compilers under the GIPC framework and the corresponding run-time environment under the eduction execution engine (GEE) among other things that communicate through the GEE Resources (GEER) (see the high-level architecture in Figure 1). These two modules are the major primary components for compilation and execution of intensional programs, which will require amendments for the changes proposed in this work of forensics.

GIPSY is also used as an investigation platform for the compilation and distributed cyberforensics evaluation of the Forensic Lucid programs [13], [12]. GEE is the component where the distributed demand-driven evaluation takes place, e.g. relying on the Demand Migration System (DMS) [36], [37], and the multi-tier architecture overall [34], [38]. The Demand Migration System (DMS) in here is an implementation of the Demand Migration Framework (DMF) introduced earlier by Vassev and extended by Pourteymour [39], [40], [41], [37]. The initial version of the DMS relied on Jini [42] for transport and storage of the demands and the results in a JavaSpaces [43] repository acting as a data warehouse cache for the most frequently demanded computations and their results (demand store). The DMF is an architecture that is centered around the demand store with transport agents (TAs) that implement a particular protocol (as a proof-of-concept Jini and JMS [44] TAs are used [40], [41]) to deliver demands between the demand store, workers (that do the actual computation for procedural demands), and generators (that request the computation to be done and collect results). Thus, GIPSY has some raw implementation of RMI [45], Jini [42], and JMS [44] and the one that relies on the DMS for its transport needs. Therefore, as-is, the distributed security aspect here solely relies on the underlying communication protocols (the bottom line is Java RMI [45]). There were proposals on how to harden GIPSY security-wise [46], but there were no any means to aid a forensic investigation of security incidents with DMS or GIPSY as a whole, if they happened to occur.

D. Distributed MARF (DMARF)

DMARF [47] is based on the classical MARF whose pipeline stages were made into distributed nodes (and later extended to be managed over SNMPv2 [48]).

Classical MARF: The Modular Audio Recognition Framework (MARF) [49], [50], [51], [52], [53] is another Java framework, and an open-source research platform and a collection of pattern recognition, signal processing, and natural language processing (NLP) algorithms written in Java and put into a modular and extensible framework facilitating addition

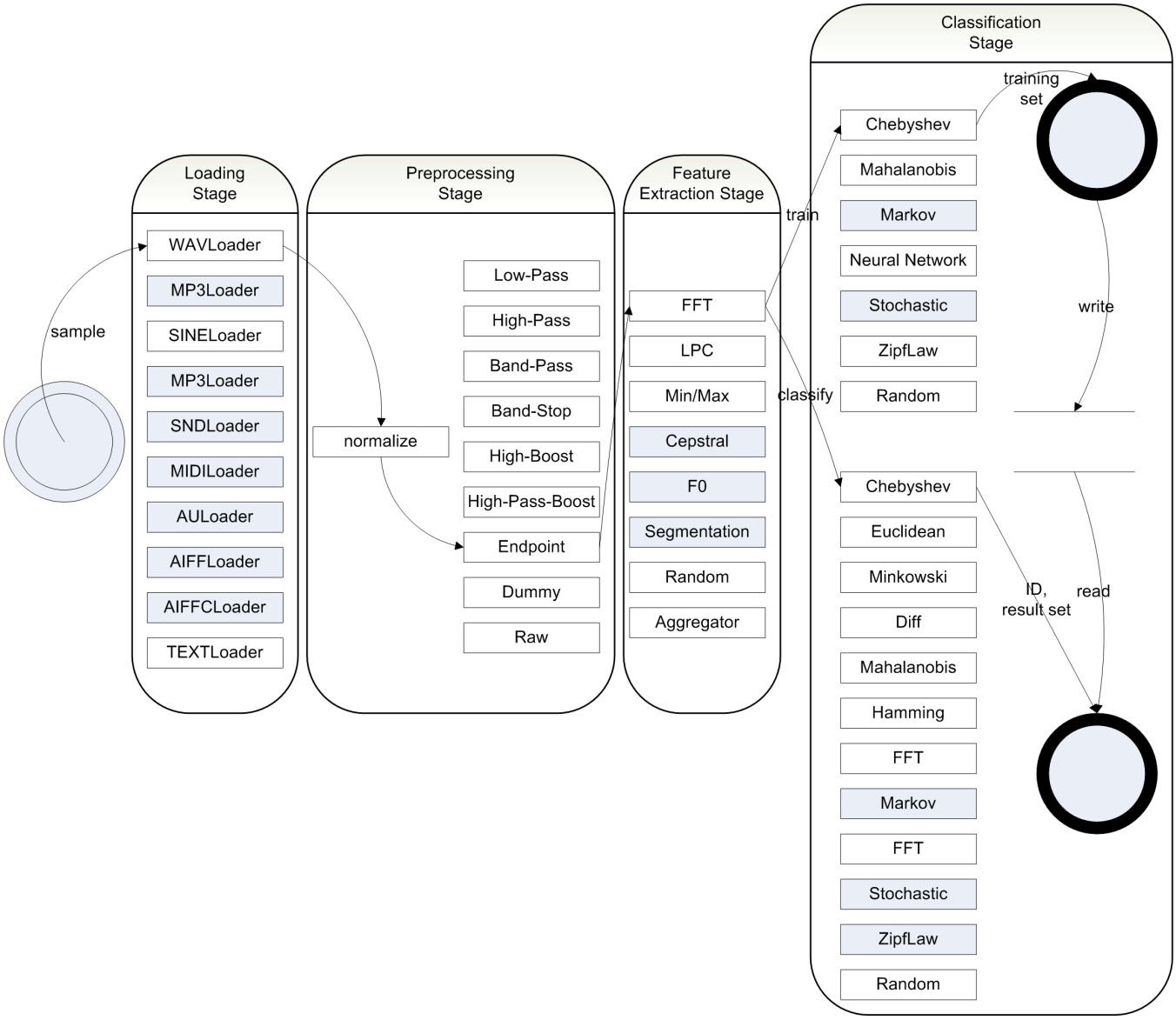

Figure 2. MARF's pattern recognition pipeline

of new algorithms for use and experimentation by scientists. A MARF instance can run over the network, run stand-alone, or may just act as a simple library in applications. The backbone of MARF consists of pipeline stages that communicate with each other to get the data they need in a chained manner. MARF's pipeline of algorithm implementations is illustrated in Figure 2, where the implemented algorithms are in white boxes, and the stubs or in-progress algorithms are in gray. The pipeline consists of four basic stages: sample loading, preprocessing, feature extraction, and training/classification.

There are a number of applications that test MARF's functionality and serve as examples of how to use MARF's modules. One of the most prominent applications is `SpeakerIdentApp` – Text-Independent Speaker Identification (who, gender, accent, spoken language, etc.) [54]. Its derivative, `FileTypeIdentApp`, was used to employ MARF's capabilities for forensic analysis of file types [55] as opposed to the Unix `file` utility [56], [57].

Classical MARF's Evidence: The evidence extracted from the analysis results of MARF comes from the several internal data structures [14], namely `Result`, `ResultSet`, `TrainingSet`, and `Configuration`. The `Result` consists of tuples containing ID and $outcome$, which are the properties of a single result. The result set, `ResultSet`, is a collection of such tuples. Processed utterances (a.k.a feature vectors or clusters of `doubles`), alongside with the training file names and IDs comprise the training set data, and configuration is a collection of processing settings that led to the current results

given the training set [14]. We will use this way of modeling of the forensic context to extend it to our other case studies.

The property P is specified in the three main categories: configuration, training set, and the result set in MARF. Its observed duration is set as a default of $(1, 0)$, as the notion of real duration varies per configuration details, but we are interested in how we arrive from the given configuration and training set to the results [14] in this case. The finer-grained details, including the actual duration may be specified inside P. `observation o = P` as syntactically written, is therefore equivalent to $o = (P, 1, 0)$ as mentioned earlier. The observation sequence os is defined as a sequence of three observations, each observation per category. The observations are ordered as (1) configuration `configo`, (2) training set `tseto`, and (3) the classification result `resulto`. The meaning of this observation sequence is that given some MARF configuration settings and the existing training set, the system produces the classification result. During the training, the observation sequence is slightly different, but also has three observations: configuration, incoming sample, and the resulting training set, which are encoded accordingly as all the necessary primitives for that are already defined [14]. The complete exportable Forensic Lucid expression is a 3-observation sequence as presented in Listing 3. With the simplifying assumption, the $(1, 0)$ syntactical constructs can be dropped, only keeping the P, which in this case is a higher-order context specification, as shown in Figure 4 [14]. Such a contextual specification of MARF internals Forensic Lucid inherited from the MARFL language [58].

```
MARFos = { confo, tseto, resulto } =
{
  ([
    sample loader      : WAV [ channels: 2, bitrate: 16, encoding: PCM, f : 8000 ],
    preprocessing      : LOW-PASS-FFT-FILTER [ cutoff: 2024, windowsize: 2048 ],
    feature extraction : LPC [ poles: 40, windowsize: 2048 ],
    classification     : MINKOWSKI-DISTANCE [ r : 6 ]
  ], 1, 0),

  ([data:{[5.2,3.5,7.5],[3.6,2.5,5.5,6.5]}, files:[``/foo/bar.wav'',``/bar/foo.wav'']], 1, 0),

  ([ID:5, outcome:1.5], 1, 0)
}
```

Figure 3. Example of a three-observation sequence context exported from MARF to Forensic Lucid.

```
MARFos = { confo, tseto, resulto } =
{
  [
    sample loader      : WAV [ channels: 2, bitrate: 16, encoding: PCM, f : 8000 ],
    preprocessing      : LOW-PASS-FFT-FILTER [ cutoff: 2024, windowsize: 2048 ],
    feature extraction : LPC [ poles: 40, windowsize: 2048 ],
    classification     : MINKOWSKI-DISTANCE [ r : 6 ]
  ],

  [data:{[5.2,3.5,7.5],[3.6,2.5,5.5,6.5]}, files:[``/foo/bar.wav'',``/bar/foo.wav'']],

  [ID:5, outcome:1.5]
}
```

Figure 4. Example of a simplified three-observation sequence context exported from MARF to Forensic Lucid.

Distributed Version: The classical MARF presented earlier was extended [47] to allow the stages of the pipeline to run as distributed nodes as well as their front-ends, as shown in Figure 5 at high-level overview. The basic stages and the front-ends were designed to support, but implemented without backup recovery or hot-swappable capabilities. They only support communication over Java RMI [45], CORBA [59], and XML-RPC WebServices [60], [61]. Being distributed, DMARF has new data structures and data flow paths that are not covered by the Forensic Lucid specification of the classical MARF, so we contribute an extension in this work.

E. Java Data Security Framework (JDSF)

JDSF [62] has been proposed to allow security researches working with several types of data storage instances or databases in Java to evaluate different security algorithms and methodologies in a consistent environment. The JDSF design includes

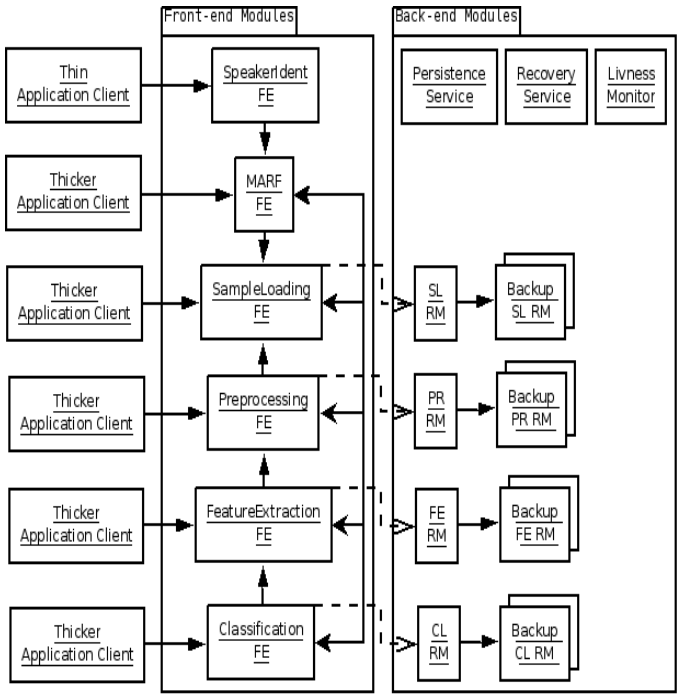

Figure 5. The distributed MARF pipeline

at the following aspects of data storage security: confidentiality (data are private), integrity (data are not altered in transit), origin authentication (data are coming from a trusted source), and SQL randomization (for relational databases only).

JDSF also provides an abstraction of the common essential cryptographic primitives. The abstraction exposes a common API proxied to the common Java open-source implementations of the cryptographic algorithms for encryption, hashing, digital signatures, etc. The higher-level JDSF design summary is illustrated in several UML specification diagrams documented in the related works [63], [64], [65], [66]. The design presented in those works illustrates all the necessary main subpackages of JDSF and its configuration, the design of security-enhanced storage, the authentication subframework, privacy subframework, integrity subframework, and the abstraction API over the concrete cryptographic primitives. JDSF is convenient to use in the scope of the research in the article, and, like GIPSY and DMARF, it is implemented in Java, and is open-source.

F. Cryptolysis

Cryptolysis [67] is a small framework that includes a collection of automated attacks on the classical ciphers using a set heuristics algorithm implementation in Java. The algorithms primarily come from the related work on the same subject [68]. Cryptolysis also features additional algorithms that are wrappers around classification and signal processing tasks of MARF [49] for additional imprecise and spectral reasoning. Cryptolysis, among other things, also performs some NLP parsing that segments the deciphered whole-block text and inserts spaces in-between the word boundaries automatically for the ease of readability. Cryptolysis, like the others, is an open-source project.

III. SELF-FORENSICS

In this section we elaborate in detail on the application of self-forensics and its requirements that must be made formal in the industry, if the property to be used in a wider context and scope.

A. Generalities

As we encode the observation sequences for our case study systems, we observe some general tendencies. It is possible to have gaps in the evidence stories when some of the expected data structures were not instantiated or some events did not occur. In this case we model them as no-observation [4], [5], \$. A *no-observation* is an observation $\$ = (C_T, 0, infinitum)$ that puts no restrictions on computations. The $infinitum$ is an integer constant that is greater than the length of any finite computation that may have happened and C is a set of all possible computations in the system T [4].

All systems' high-level evidence follows a similar traditional blackbox evidential pattern: *given an instance of configuration followed by the initial "input" data followed by a one or more computations on the data followed by a some sort of result or*

a result set. This contextual pattern can be exploded in the context level to the very fine-grained details to the bottom level of core data structures and variables. The computations can be arbitrarily complex and nesting of the context expressions, as evidential logs, can be arbitrarily deep, depending on the desired level-of-detail (LoD).

Inherently parallel evaluation can also be captured as variable length generic observations [4], [5], forming a multidimensional *box* [22] where each element can be evaluated in parallel, and all elements should complete the evaluation prior transiting to the final result or result set observation.

Given this initial discussion we further proceed with some of the contextual Forensic Lucid specifications of our studied systems, specifically DMARF in Section III-B, GIPSY in Section III-C, JDSF in Section III-D, and Cryptolysis in Section III-E.

B. DMARF

Distributed aspect of DMARF, as opposed to the classical MARF, makes gathering of more forensic data due to intrinsically more complex communication between modules, and potentially remote ones. In addition to the earlier specification recited in Figure 4, we need to capture the configuration data related to the connection settings, protocols, and any other properties related to the distributed computing.

The main difference is that now given the same configuration it is possible to have multiple distributed/parallel training sets to be computed as well as multiple results produced on different nodes and the configuration object "explosion" will include the communication protocols and the related information. In Equation 4 is a complete high-level specification of a DMARF run of observations. $dconfo$ is an observation of the inital DMARF configuration, $tseto_i$ is the observation of the $X \times Y$ training sets, where X and Y form a *box* of all possible training sets that can be available in that run, and the $resulto_i$ is an observation of possible corresponding results of the pipelines' runs. There could be multiple pipelines formed through the lifetime of a DMARF network of the computing stages and nodes.

$$(dconfo, 1, 0), (tseto_i, X, Y), \ldots, (resulto_i, X, Y) \tag{4}$$

A system instance that has not produced a result (e.g. due to a crash at the classification stage) would still have a 3-observation sequence, with the last one being a *no-observation*, as shown in Equation 5.

$$(dconfo, 1, 0), (tseto_i, X, Y), \ldots, \$ \tag{5}$$

It is theoretically possible to have no-observations for the configuration or training set data components say in case such real evidential data was lost or is not trustworthy due to poor handling or improper chain of custody. In such a case only partial reasoning can be performed on the evidence.

Note, at this point we do not cover the typical advanced distributed systems features such as the replication, write-ahead-logging, and load balancing and their related data structures, communication, etc. that deserve a separate large publication and for now focus only on the normal "business" operation of the system.

C. GIPSY

In GIPSY, the core data structure, connecting the compiler GIPC and the execution engine GEE is the GEER, represented by the `GIPSYProgram` class. This is the initial observation as far as GEE concerned. It causes the engine to produce a number of demands in the order of the abstract syntax tree (AST) traversal, abstracted by the `Demand` class and the corresponding `IDemand` interface. Demands live and die inside a some sort of store, that acts like a cache and are transported by the transport agents (TAs) using various communication technologies mentioned earler in the previous section. `IDemand` is used for both demands and their results. When the result of computation reaches the top of the tree resolution, the final result of the program is computed – it is an instance of some subclass of the `GIPSYType` from the GIPSY Type System [69]. Thus, the high-level observation sequence is encoded as:

$$(p, 1, 0), (d_i, X, Y), \ldots, (r, 1, 0) \tag{6}$$

where `GIPSYProgram` p – is the initial program (GEER), `IDemand` d_i – is a cross product $X \times Y$ of all possible demands that may be observed, `GIPSYType` r – is the result of computation. X and Y form a cross product of all possible demands that can be made throughout the program execution, of whether they need to be computed or retrieved from a cache, an intensional data warehouse. An example of the observation sequence expression is in Figure 6. The `AST` corresponds to the abstract syntax tree of a compiled GIPSY program, followed by a dictionary of identifiers and the format of the compiled language. Then, at run-time demands are generated for the two identifiers at the dimension $d : 1$ as well as their sum with the $+$ operator yielding a resulting value r.

```
GIPSYos = { p, d1, d2, di, r } =
{
  (
    [
      AST        : ..., // abstract syntax tree
      dictionary : [id1:'x', id2:'y'],
      formattag  : COMPILED_GEER
    ], 1, 0
  ),

  ([identifier:id1, tag:d, tagval:1], 1, 0),
  ([identifier:id2, tag:d, tagval:1], 1, 0),
  ([identifier:+, tag:d, tagval:1], 1, 0),

  ([result:X], 1, 0)
}
```

Figure 6. Example of observations in GIPSY.

The high-level encoding does not constitute the compilation data structures used in the GIPC compiler family at this point as we are more concerned with the run-time execution at this point; however, one has to plan for certified compilation eventually, which is becoming more important these days, and that's what we will recommend to the GIPSY designers and developers for the future work. Once certified compilation is in place, the corresponding compilation evidence can be encoded for further offline or run-time analysis. Likewise, as in DMARF, we avoid talking about typical distributed system properties that ensure availability of the services, replication, etc. but in the final self-forensics modules design all the distributed middleware also needs to be covered.

D. JDSF

JDSF's core data structures comprise secure beans amended with security information layered as needed [62]. Thus, most of the observations comprise these data structures, the beans, and the observation sequence, using strict ordering gives away the sequence in which a given security information was applied, e.g. signing happened before encryption and so on. Another observation is the configuration object instance that provides the details of all the module configuration parameters. The final observation is the secure object after application of all the security levels. Thus, the high-level sequence of events captures the observation context of given data and configuration that result in a secured bean. Thus, the data structures covered by the Forensic Lucid context include the base `SecureBean` object type, and its derivatives `AuthenticatedObject`, `EncryptedObject`, `IntegrityAddedObject`. The basic forensic context structure is, therefore:

```
bean: { value:X, security_info:Y }
```

The dimensions `value` and `security_info` comprise the concrete security-enhanced payload data X and the details about the said security applied Y. On top of the actual beans, we have the corresponding configuration object that is a part of the system's context. The security `Configuration` object's context varies in three basic dimensions:

```
configuration:
[
  confidentiality:C,
  integrity:I,
  authentication:A
]
```

These dimension's tag values evaluate to integers enumerating concrete algorithms used for each of these stages. The context also determines in which order they were applied, which may be an important factor e.g. in the public key encryption schemes where the order of signing and encryption is important. In Figure 7 is an example of the Forensic Lucid context instance of a particular configuration.

```
JDSFos = { config, data, bean } =
{
  (
    ordered [
      confidentiality : RSA  [ key:12345, keylength:1024 ],
      integrity       : MD5,
      authentication  : DSA  [ key:23456, keylength:256 ]
    ], 1, 0
  ),

  ([1,2,3,4], 1, 0),

  ([value:[3,1,4,2], security_info:Y], 1, 0)
}
```

Figure 7. Example of observations in JDSF.

E. Cryptolysis

Cryptolysis being a small system does not have a lot of deep contextual complexity. It has no explicit configuration object at this point and it is uncertain if it's going to have one, but there are implicit configuration settings that impact the results of the runs of the systems. There there are key several operational evidential context pieces: the cryptographic `Key`, the plain-text and cipher-text data, cipher algorithm type, cryptanalysis algorithm type, statistics, and a result. The `Key` k in the forward encryption process is a part of input configuration, along with the cipher algorithm to use r, and the input plain text data t, and the result is the cipher text c. In the reverse process, which is the main mode of operation for Cryptolysis, the configuration consists of the cryptanalysis algorithm type a, followed by the input cipher text data c, and the primary result is the guessed encryption `Key` k and the secondary result is the associated accuracy statistics s built along, and the plain text t, as in Equation 7 and Equation 8. The two equations denote these two scenarios; the latter is exemplified in Figure 8.

$$(k, 1, 0), (r, 1, 0), (t, 1, 0), (c, 1, 0) \tag{7}$$

$$(a, 1, 0), (c, 1, 0), (k, 1, 0), (s, 1, 0), (t, 1, 0) \tag{8}$$

```
Cryptolysis_os = { a, c, k, s, t } =
{
  ([alogrithm:GENETIC_ALGORITHM], 1, 0),
  (['k','h','o','o','r'], 1, 0)
  ([key:[xyzabc...], length:26], 1, 0),
  ([accuracy:0.95], 1, 0),
  (['h','e','l','l','o'], 1, 0)
}
```

Figure 8. Example of observations for the second equation in Cryptolysis.

There is also a natural language processing (NLP) aspect of Cryptolysis that we do not cover in our work here that deals with natural language word boundary detection in the deciphered text that has no punctuation or spaces.

F. Summary

After having specified the high-level evidential observation sequences, each self-forensic module designer need to go and specify the "gory" details, as needed and deemed important. Then, following the normal operation of the systems the self-forensic modules are turned on and logging data in the Forensic Lucid format continuously (notice, the forensic logs of course themselves must of of high integrity and confidentiality, and there should be enough external storage to accommodate them, but these problems are not specific to this research, are addressed for other needs as well elsewhere like any type of

logging or continuous data collection). The format, assuming being a standard, would be processable by Forensic Lucid-enabled expert systems, fully automatically (good for autonomous systems) or interactively with the investigators making their case and analyzing the gathered evidence. Making a step further, such approach can be generalized and put outside of the cybercrime domain into the autonomous transport vehicle design and other robotic systems and spacecraft to help investigating engineering teams to investigate and troubleshoot the systems and investigate crashes and other incidents with the event reconstruction.

IV. Conclusion

Such forensic specification is not only useful for cybercrime investigations, incident analysis, and response, but also useful to train new software engineers on a team, and others involved, in data analysis, potentially overlooking data and making incorrect ad-hoc decisions. In a Forensic Lucid-based expert system one can accumulate a number of contextual facts from the self-forensic evidence and the trainees can construct their theories of what happened and see of their theories agree with the evidential data. Over time, one can potentially accumulate the general enough contextual knowledge base of encoded facts that can be analyzed across cases, globally and on the web. Such data can be shared across law enforcement, military, and research institutions.

We note that all studied systems are remarkably similar in their evidence structure suggesting that a reusable model can be built to observe a large number of similar systems for self-forensic evidence.

Using an AspectJ-based implementation and component- or even method-level external observations can give very fine-grained as well as coarse-grained observation sequences and allow to model not only the contexts in the form of observation sequences of data structures and data flows, but also *automatically* build the state transition function ψ in Forensic Lucid to be ready to be exported for the use by investigators to allow modeling the Ψ^{-1} for event reconstruction. This approach can be very well used for automated debugging of complex systems allowing the developers and quality assurance teams trace the problem and recreate the sequence of events back to what may have been the cause better than a mere stack trace and the resulting core dump. Moreover, mobile devices these days increasingly use Java, so the said forensic data can be created on-the-fly with AspectJ observation of mobile device OS components.

V. Future Work

As a part of the future work in this direction we plan to complete the implementation of the self-forensics modules to generate evidence according to the specifications presented earlier. Specifically, our priority is to implement the notion of self-forensics in the GIPSY [70], [32], [34], [71] and DMARF [61], [72], [73], [74] systems. Then, we plan to gather performance and storage overhead statistics when the self-forensics modules are turned on. We also need to improve the granularity of the event collection, correlation, and encoding with AspectJ-based tracing on the level of the method calls with the *before* and *after* triggers. Further, we want to explore events with probabilities, credibility, trustworthiness factors and the like using a probabilistic approach when exact reasoning is not possible.

In another direction of the research and validation of the approach for the software systems, we plan to amend the Autonomic Systems Specification Language (ASSL) and the corresponding set of tools [75], [76], [77], [78], [79] to handle the self-forensics property.

We also need to explore and specify on how malicious and determined attackers would be dealt with by the self-forensics methods and techniques to retain valuable trustworthy forensic evidence. (In the nutshell, the issue is similar to remove logging in the specified format, perhaps offsite, maintaining a secure channel, but this deserves a separate complete discussion; additionally an investigated concept, like self-protection for DMARF [73] can be employed as one of the assurance techniques for self-forensics).

Acknowledgments

This work was supported in part by NSERC, the Faculty of Engineering and Computer Science of Concordia University, Montreal, Quebec, Canada and by an IRCSET postdoctoral fellowship grant (now termed as EMPOWER) at University College Dublin, Ireland, by the Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre.

REFERENCES

[1] S. A. Mokhov, "The role of self-forensics in vehicle crash investigations and event reconstruction," [online], May 2009, http://arxiv.org/abs/0905.2449.

[2] ——, "Towards improving validation, verification, crash investigations, and event reconstruction of flight-critical systems with self-forensics," [online], Jun. 2009, http://arxiv.org/abs/0905.2449.

[3] Wikipedia, "S.M.A.R.T. — Wikipedia, the free encyclopedia," [Online; accessed 9-February-2009], 2009, http://en.wikipedia.org/w/index.php?title=S.M.A.R.T.&oldid=269322389.

[4] P. Gladyshev and A. Patel, "Finite state machine approach to digital event reconstruction," *Digital Investigation Journal*, vol. 2, no. 1, 2004.

[5] P. Gladyshev, "Finite state machine analysis of a blackmail investigation," *International Journal of Digital Evidence*, vol. 4, no. 1, 2005.

[6] A. R. Arasteh and M. Debbabi, "Forensic memory analysis: From stack and code to execution history," *Digital Investigation Journal*, vol. 4, no. 1, pp. 114–125, Sep. 2007.

[7] A. R. Arasteh, M. Debbabi, A. Sakha, and M. Saleh, "Analyzing multiple logs for forensic evidence," *Digital Investigation Journal*, vol. 4, no. 1, pp. 82–91, Sep. 2007.

[8] M. Debbabi, A. R. Arasteh, A. Sakha, M. Saleh, and A. Fry, "A collection of JPF forensic plug-ins," Computer Security Laboratory, Concordia Institute for Information Systems Engineering, 2007–2008.

[9] R. Hadjidj, M. Debbabi, H. Lounis, F. Iqbal, A. Szporer, and D. Benredjem, "Towards an integrated e-mail forensic analysis framework," *Digital Investigation*, vol. 5, no. 3-4, pp. 124–137, 2009.

[10] C. Gundabattula and V. G. Vaidya, "Building a state tracing Linux kernel," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, O. Göbel, S. Frings, D. Günther, J. Nedon, and D. Schadt, Eds., Mannheim, Germany, Sep. 2008, pp. 173–196, LNI140.

[11] AspectJ Contributors, *AspectJ: Crosscutting Objects for Better Modularity*. eclipse.org, 2007, http://www.eclipse.org/aspectj/.

[12] S. A. Mokhov, J. Paquet, and M. Debbabi, "Formally specifying operational semantics and language constructs of Forensic Lucid," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, O. Göbel, S. Frings, D. Günther, J. Nedon, and D. Schadt, Eds. Mannheim, Germany: GI, Sep. 2008, pp. 197–216, LNI140.

[13] S. A. Mokhov and J. Paquet, "Formally specifying and proving operational aspects of Forensic Lucid in Isabelle," Department of Electrical and Computer Engineering, Concordia University, Tech. Rep. 2008-1-Ait Mohamed, Aug. 2008, in Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[14] S. A. Mokhov, "Encoding forensic multimedia evidence from MARF applications as Forensic Lucid expressions," in *Proceedings of CISSE'08*. University of Bridgeport, CT, USA: Springer, Dec. 2008, to appear.

[15] M. Spichkova, "Focus on Isabelle: From specification to verification," Department of Electrical and Computer Engineering, Concordia University, Tech. Rep. 2008-1-Ait Mohamed, Aug. 2008, in Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[16] W. Wadge and E. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[17] E. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge, *Multidimensional, Declarative Programming*. London: Oxford University Press, 1995.

[18] E. A. Ashcroft and W. W. Wadge, "Lucid - a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 5, no. 3, 1976.

[19] ——, "Erratum: Lucid - a formal system for writing and proving programs." *SIAM J. Comput.*, vol. 6, no. (1):200, 1977.

[20] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge, "Sequential demand-driven evaluation of eager TransLucid," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008, pp. 1266–1271.

[21] J. Paquet, S. A. Mokhov, and X. Tong, "Design and implementation of context calculus in the GIPSY environment," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008, pp. 1278–1283.

[22] K. Wan, "Lucx: Lucid enriched with context," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[23] X. Tong, "Design and implementation of context calculus in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2008.

[24] R. Murch, *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall, 2004.

[25] M. Parashar and S. Hariri, Eds., *Autonomic Computing: Concepts, Infrastructure and Applications*. CRC Press, Dec. 2006.

[26] H. Hiss, "Checking the satisfiability of XML-specifications," Department of Electrical and Computer Engineering, Concordia University, Tech. Rep. 2008-1-Ait Mohamed, Aug. 2008, in Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[27] The PRISM Team, "PRISM: a probabilistic model checker," [online], 2004–2009, http://www.prismmodelchecker.org/, last viewed June 2009.

[28] Y. M. Ding, "Bi-directional translation between data-flow graphs and Lucid programs in the GIPSY environment," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[29] J. Paquet, "Scientific intensional programming," Ph.D. dissertation, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[30] J. Paquet and P. Kropf, "The GIPSY architecture," in *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[31] B. Lu, P. Grogono, and J. Paquet, "Distributed execution of multidimensional programming languages," in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, vol. 1. International Association of Science and Technology for Development, Nov. 2003, pp. 284–289.

[32] J. Paquet and A. H. Wu, "GIPSY – a platform for the investigation on intensional programming languages," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. Las Vegas, USA: CSREA Press, Jun. 2005, pp. 8–14.

[33] S. A. Mokhov, "Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005, ISBN 0494102934.

[34] J. Paquet, "A multi-tier architecture for the distributed eductive execution of hybrid intensional programs," in *Proceedings of 2nd IEEE Workshop in Software Engineering of Context Aware Systems (SECASA'09)*. IEEE Computer Society, 2009, to appear.

[35] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Communication of the ACM*, vol. 20, no. 7, pp. 519–526, Jul. 1977.

[36] E. I. Vassev, "General architecture for demand migration in the GIPSY demand-driven execution engine," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Jun. 2005, ISBN 0494102969.

[37] A. H. Pouteymour, "Comparative study of Demand Migration Framework implementation using JMS and Jini," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2008.

[38] B. Han, S. A. Mokhov, and J. Paquet, "Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java," [online], 2009, http://arxiv.org/abs/0906.4837.

[39] E. Vassev and J. Paquet, "A generic framework for migrating demands in the GIPSY's demand-driven execution engine," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. Las Vegas, USA: CSREA Press, Jun. 2005, pp. 29–35.

[40] A. H. Pourteymour, E. Vassev, and J. Paquet, "Towards a new demand-driven message-oriented middleware in GIPSY," in *Proceedings of PDPTA 2007*, PDPTA. Las Vegas, USA: CSREA Press, Jun. 2007, pp. 91–97.

[41] ——, "Design and implementation of demand migration systems in GIPSY," in *Proceedings of PDPTA 2009*. Las Vegas, USA: CSREA Press, Jun. 2008.

[42] Jini Community, "Jini network technology," [online], Sep. 2007, http://java.sun.com/developer/products/jini/index.jsp.

[43] Q. H. Mamoud, "Getting started with JavaSpaces technology: Beyond conventional distributed programming paradigms," [online], Jul. 2005, http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/.

[44] Sun Microsystems, "Java Message Service (JMS)," [online], Sep. 2007, http://java.sun.com/products/jms/.

[45] A. Wollrath and J. Waldo, "Java RMI tutorial," Sun Microsystems, Inc., 1995–2005, http://java.sun.com/docs/books/tutorial/rmi/index.html.

[46] S. A. Mokhov, "Towards security hardening of scientific distributed demand-driven and pipelined computing systems," in *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*. Krakow, Poland: IEEE Computer Society, Jul. 2008, pp. 375–382.

[47] ——, "On design and implementation of distributed modular audio recognition framework: Requirements and specification design document," [online], Aug. 2006, project report, http://arxiv.org/abs/0905.2459, last viewed May 2009.

[48] S. A. Mokhov, L. W. Huynh, and J. Li, "Managing distributed MARF's nodes with SNMP," in *Proceedings of PDPTA'2008*, vol. II. Las Vegas, USA: CSREA Press, Jul. 2008, pp. 948–954.

[49] The MARF Research and Development Group, "The Modular Audio Recognition Framework and its Applications," SourceForge.net, 2002–2009, http://marf.sf.net, last viewed December 2008.

[50] S. Mokhov, I. Clement, S. Sinclair, and D. Nicolacopoulos, "Modular Audio Recognition Framework," Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2003, project report, http://marf.sf.net, last viewed April 2008.

[51] S. A. Mokhov, "Introducing MARF: a modular audio recognition framework and its applications for scientific and software engineering research," in *Advances in Computer and Information Sciences and Engineering*. University of Bridgeport, U.S.A.: Springer Netherlands, Dec. 2007, pp. 473–478, proceedings of CISSE/SCSS'07, cisse2007.org.

[52] ——, "Choosing best algorithm combinations for speech processing tasks in machine learning using MARF," in *Proceedings of the 21st Canadian AI'08*, S. Bergler, Ed. Windsor, Ontario, Canada: Springer-Verlag, Berlin Heidelberg, May 2008, pp. 216–221, LNAI 5032.

[53] ——, "Study of best algorithm combinations for speech processing tasks in machine learning using median vs. mean clusters in MARF," in *Proceedings of C3S2E'08*, B. C. Desai, Ed. Montreal, Quebec, Canada: ACM, May 2008, pp. 29–43, ISBN 978-1-60558-101-9.

[54] S. A. Mokhov, S. Sinclair, I. Clement, D. Nicolacopoulos, and the MARF Research & Development Group, "Text-Independent Speaker Identification Application," Published electronically within the MARF project, http://marf.sf.net, 2002-2008, last viewed April 2008.

[55] S. A. Mokhov and M. Debbabi, "File type analysis using signal processing techniques and machine learning vs. `file` unix utility for forensic analysis," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, O. Goebel, S. Frings, D. Guenther, J. Nedon, and D. Schadt, Eds. Mannheim, Germany: GI, Sep. 2008, pp. 73–85, LNI140.

[56] I. F. Darwin, J. Gilmore, G. Collyer, R. McMahon, G. Harris, C. Zoulas, C. Lowth, E. Fischer, and Various Contributors, "`file` – determine file type, BSD General Commands Manual, file(1)," BSD, Jan. 1973–2007, man file(1).

[57] ——, "`file` – determine file type," [online], Mar. 1973–2008, ftp://ftp.astron.com/pub/file/, last viewed April 2008.

[58] S. A. Mokhov, "Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008, pp. 1288–1294.

[59] Sun Microsystems, "Java IDL," Sun Microsystems, Inc., 2004, http://java.sun.com/j2se/1.5.0/docs/guide/idl/index.html.

[60] ——, "The java web services tutorial (for Java Web Services Developer's Pack, v2.0)," Sun Microsystems, Inc., Feb. 2006, http://java.sun.com/webservices/docs/2.0/tutorial/doc/index.html.

[61] S. A. Mokhov and R. Jayakumar, "Distributed modular audio recognition framework (DMARF) and its applications over web services," in *Proceedings of TeNe'08*. University of Bridgeport, CT, USA: Springer, Dec. 2008, to appear.

[62] S. A. Mokhov, L. W. Huynh, J. Li, and F. Rassai, "A Java Data Security Framework (JDSF) for MARF and HSQLDB," Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada, Apr. 2007, project report. Hosted at http://marf.sf.net, last viewed April 2008.

[63] ——, "A privacy framework within the java data security framework (JDSF): Design refinement, implementation, and statistics," in *Proceedings of the 12th World Multi-Conference on Systemics, Cybernetics and Informatics (WM-SCI'08)*, N. Callaos, W. Lesso, C. D. Zinn, J. Baralt, J. Boukachour, C. White, T. Marwala, and F. V. Nelwamondo, Eds., vol. V. Orlando, Florida, USA: IIIS, Jun. 2008, pp. 131–136.

[64] S. A. Mokhov, L. W. Huynh, and L. Wang, "The integrity framework within the java data security framework (JDSF): Design refinement and implementation," in *Proceedings of CISSE'08*. University of Bridgeport, CT, USA: Springer, Dec. 2008, to appear.

[65] S. A. Mokhov, F. Rassai, L. W. Huynh, and L. Wang, "The authentication framework within the java data security framework (JDSF): Design refinement and implementation," in *Proceedings of CISSE'08*. University of Bridgeport, CT, USA: Springer, Dec. 2008, to appear.

[66] S. A. Mokhov, J. Li, and L. Wang, "Simple dynamic key management in SQL randomization," 2009, submitted for publication to CCT'09.

[67] S. A. Mokhov, M.-A. Laverdière, N. Hatami, and A. Benssam, "Cryptolysis v.0.0.1 - A Framework for Automated Cryptanalysis of Classical Ciphers," Unpublished, 2005, project report.

[68] A. Clark and E. Dawson, "Optimisation heuristics for the automated cryptanalysis of classical ciphers," *Journal of Combinatorial Mathematics and Combinatorial Computing*, 1998.

[69] S. A. Mokhov, J. Paquet, and X. Tong, "A type system for hybrid intensional-imperative programming support in GIPSY," in *Proceedings of C3S2E'09*. New York, NY, USA: ACM, May 2009, pp. 101–107.

[70] E. Vassev and J. Paquet, "Towards autonomic GIPSY," in *Proceedings of the Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASE 2008)*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 25–34.

[71] S. A. Mokhov and J. Paquet, "Using the General Intensional Programming System (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions," [online], Jun. 2009, http://arxiv.org/abs/0906.3911.

[72] E. Vassev and S. A. Mokhov, "Towards autonomic specification of Distributed MARF with ASSL: Self-healing," 2009, submitted for publication to Middleware'09.

[73] S. A. Mokhov and E. Vassev, "Autonomic specification of self-protection for Distributed MARF with ASSL," in *Proceedings of C3S2E'09*. New York, NY, USA: ACM, May 2009, pp. 175–183.

[74] E. Vassev and S. A. Mokhov, "Self-optimization property in autonomic specification of Distributed MARF with ASSL," in *Proceedings of ICSOFT'09*. Sofia, Bulgaria: INSTICC, Jul. 2009, to appear.

[75] E. Vassev and J. Paquet, "ASSL – Autonomic System Specification Language," in *Proceedings if the 31st Annual IEEE / NASA Software Engineering Workshop (SEW-31)*, NASA/IEEE. Baltimore, MD, USA: IEEE Computer Society, Mar. 2007, pp. 300–309.

[76] ——, "Towards an autonomic element architecture for ASSL," in *Proceedings of the 29th IEEE International Conference on Software Engineering / Software Engineering for Adaptive and Self-managing Systems (ICSE 2007 SEAMS)*. Minneapolis, MN, USA: IEEE, May 2007, p. 4.

[77] E. Vassev, H. Kuang, O. Ormandjieva, and J. Paquet, "Reactive, distributed and autonomic computing aspects of AS-TRM," in *Proceedings of the 1st International Conference on Software and Data Technologies - ICSOFT'06*, 2006, pp. 196–202.

[78] E. I. Vassev, "Towards a framework for specification and code generation of autonomic systems," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[79] E. Vassev and S. A. Mokhov, "An ASSL-generated architecture for autonomic systems," in *Proceedings of C3S2E'09*. New York, NY, USA: ACM, May 2009, pp. 121–126.